CORES 64

EDITOR/ASSEMBLER & MONITOR

MICRODEAL

CO-RESIDENT EDITOR/ASSEMBLER
STARTUP PROCEDURES

CORES-64 is an M6809 processor machine language program on a Dragon computer formatted machine language tape. To load the program into the computer type CLOADM and press ENTER. The screen should clear and the tape should start to load. When the program is completely loaded and no load errors were detected CORES-64 will automatically start execution and will select 64K if the extra memory is installed. When the initialization process is completed the program should display a message and then the READY prompt. You are now in the Editor program and ready to begin editing or creating a new text file. The system is now completely under the control of CORES-64 so don't expect it to respond to any of "BASICs" commands. In order to return control to "BASIC" CORES-64 must first be exited by the command "EXIT". Once exited you cannot return to CORES-64.

CORES-64 has the ability to load BASIC program tape files provided they were saved in an ASCII format. BASIC can save program files in this manner, but it normally saves them in a binary form. In order to cause the file to be in a compatible format with the editor you should add an "A" to the end of the normal BASIC CSAVE command. CSAVE "PROGRAM",A (enter). This would tell BASIC to save a file with the name "PROGRAM" in an ascii format. When reloading the file, BASIC automatically can tell what type of file it is and load it accordingly. All files created by CORES-64 and saved to tape are in an ASCII format which is compatible with the BASIC file format. In order to try this out and may be to become more familiar with the editor, load the demonstration program "DEMO$" which is on the CORES-64 tape immediately after CORES-64. To load the file just type "TLOAD" (enter) and the file will load automatically. When the READY prompt appears the file is in the text buffer and ready for you to edit. Try several of the commands in the editor in order to become familiar with it's commands.

A great deal of time and effort has gone into the making of CORES-64. In spite of this it is possible that this software may contain some bugs. If you should encounter a problem please let us know about it in writing with sufficient information to recreate the problem. A solution will be sent to all purchasers of this program if a reasonable solution can be found.

COMMAND

INDEX

Assembler Commands

Text Editor Commands

INSTRUCTIONS TEXT EDITOR

DEFINITIONS

"\" - is the character displayed when the SHIFT & "@" keys are depressed as a delimiter for the "SEARCH" & "REPLACE" commands. Also see editor command summary.

"()" - items enclosed within these characters are required by that command to perform correctly.

"[]" - items enclosed within these characters are considered as optional, when used they must be in the required order.

"<>" - items enclosed within these characters are comments

Enter is used to denote "ENTER" character and is used to signify the completion of a line entry.

"-" - "Dash" is used as a delimiter between line numbers

"←" - Left arrow is recognised as a Backspace

"BREAK" - is used for Break control at any time to return to "READY"

Any key can be used to stop the present output and it will be resumed upon entry of any key but "BREAK". All commands can be abbreviated by using the first two characters of the command followed by its normal parameters.

LINE ENTRY:

Enter a line number, followed by a space and text ending with "Enter". Line Entry in the editor is identical to the format that Basic uses for program lines, only line numbers can be a maximum of 4 digits long.

The line buffer is preset to 80 characters and the cursor will not advance past the last character position, nor willit backspace beyond the first character position. Line length may be changed by the LL command. Fifteen characters before the end of line a medium tone beep will be heard and a higher tone beep will be heard at the end of line, if you need to continue to complete a word hit the "Clear" key for margin release. Every key you hit after that will produce a medium tone beep until you hit the enter key or reach the maximum line length of 255 at which time the EOL beep will be heard. Any time during line entry if an invalid control character key is entered a double low tone beep will be heard.

Entry of a line number over four digits will result in only the last four digits being accepted.

Entry of a line number followed by "ENTER" will delete the line previously entered using that line number the same as Basic does with program lines.

Entry of a new line using a previously entered line number will cause that line to be replaced with the new line, the same as Basic does when changing a program line.

Entry of a line with a line number between two previously entered line numbers will insert the new line between them. Once again this is identical to the way Basic inserts lines in a program.

LIST [line number] (-) [line number]

Entry without line numbers will list the entire file. Entry with a single line number will list only that line. Entry of two line numbers will list from the first line number to the second one. This is very similar to the "Basic" list function

Example: LIST 100-300 <Enter>

NLINES <COMMAND LINE>

Causes line number printing to be supressed until command following. This can be useful for printer listings where line numbers are not wanted as in letters. It can be used with almost all commands.

Example: NLINES LIST 100-300 <Enter>

RESEQUENCE [1 digit increment] [starting line#]

Causes the file to be renumbered, if no increment is specified a value of 10 is used. If no starting line # specified, line # 0000 plus the increment value is used.

Example: RESEQUENCE 5 100

Resequence the line numbers in the file begin with '100' and increment each line number by '5'.

DELETE <begin line#>-<end line#>

The delete function allows large segments of the text buffer to be removed without having to enter each line number to be deleted. If no line specifications are entered the user will be prompted as to whether the entire contents of the buffer are to be deleted, this is mainly to prevent the accidental deletion of the text buffer contents.

Example: DELETE 100-199 <Enter>

Remove all the lines in the text buffer between and including line 100 thru 199.

RLINES <Enter>

Remove all line numbers from the text buffer. This function will shorten the file by a substantial amount (5 characters per line) and make the file compatible with other editors & programs. It will save a substantial amount of space when the file is saved on tape and can be processed by the assembler in this form. However line numbers must be re-inserted by the ILINES command before any editing can be performed on the text buffer.

ILINES <Enter>

Insert line numbers on each line of text in the buffer. This will replace the line numbers previously removed by the 'RLINES' command or insert line #s in files previously created by other editors. This command will place four digits and a space on the front of each line in the highest available sequence.

LLNGTH <value>

The input buffer line length normally allows line lengths up to 128 characters to be entered by default. The LLngth command allows this value to be changed to any value from 1-255. If you are working with an 80 column printer for instance and doing text or letter writing it can be more convenient to limit the line length to the same length as the output device.

Example: LLNGTH 80 <Enter>

TLOAD (file name) <Enter>

This is the tape file load command and is used to load "ASCII" formatted "BASIC" files or files previously saved by the editor. The file name can be omitted and if so it will attempt to load the next file on the tape. If an error should occur or an attempt to load an invalid file type an error message will be displayed and the load aborted.

Example: TLOAD TEXT1 <Enter>

TSAVE (file name) <Enter>

The TSAVE command is used to save the contents of the current text buffer in an "ASCII" formatted tape file. Here again the file name can be omitted but is recommended for ease of file identification. The output file is fully compatible with the "BASIC" tape format and can be reloaded with the BASIC "CLOAD" command.

Example: TSAVE TEXT1 <Enter>

TAPPEND (file name) <Enter>

The APPEND command allows a tape file to be appended to the end of the current text file in memory. The file name can be omitted and if so it will attempt to load the next file in the tape.

Example: TAPEND TEXT2 <Enter>

SKIP <file name>

This command will allow the editor to search for and skip over tape files much the same as BASIC does. If an error is encountered while reading a file, an error message wil be displayed and the tape stopped. This can be useful for checking tape files as well as positioning tape for file additions. If no file name is used it will simply skip the next file on the tape.

Example: SKIP DEMO$ <Enter>

This would tell the editor to skip the file DEMO$

SEARCH [line#] (-) [line#] \ [string] \

Searches for all occurrences of the string between the delimiters (Shift @) and the lines containing it will be output. If the optional start & stop line is omitted the search will begin at the beginning of the file to the end of the file. If only the start line# is specified it will search to the end of file.

Example: SEARCH 100-199\TEST \

List all the lines containing the string "TEST" between lines 100 thru 199.

**RPLACE [line#] (-) [line#] \ [string] \ [string] **

This function will replace all occurrences of the first string between delimiters with the second string. If the optional line#'s are not specified the entire file will be used, if only the starting line # is specified only from there to the end of file will be used, and if both start and end line #'s are specified only the lines including them will be used.

Example: RPLACE 100-999 \TEST\TESTER\

This would tell the editor to replace all occurrences of "TEST" between lines 100 and 999 with "TESTER"

LEDIT [line#]

Causes the line number specified to be displayed and the cursor to be positioned under the first character of the line. The EDIT mode is then entered, see edit functions under "AEDIT".

Example: LEDIT 110 <Enter>

Edit line number 100 using the edit functions.

AEDIT [line#]

Causes the automatic edit mode to be entered, if the starting line # is specified the edit function will continue from that line until the end or a cancel edit operation character is entered. All the edit commands are the same as LEDIT (line edit). If no change is required on a line, enter a LF and the next line will be brought up for editing. If the line is to be deleted just enter Shift "@"

Example: AEDIT 100 <Enter>

Begin automatic line editing starting at line 100.

EDIT FUNCTION KEYS

FUNCTION	DEPRESS
MOVE CURSOR RIGHT	Right arrow key
MOVE CURSOR RIGHT 1 WORD	Clear key
MOVE CURSOR LEFT (backspace)	Left Arrow key
INSERT SPACE	Shift & Up arrow keys
DELETE CHARACTER	Shift & Down arrow keys
MOVE CURSOR TO END OF LINE	Shift & Right arrow keys
MOVE CURSOR TO BEGIN OF LINE	Shift & Left arrow keys
GOTO NEXT SEQUENTIAL LINE	Down arrow key
GOTO PREVIOUS LINE	Up arrow key
END LINE AT CURSOR POSITION	Shift & @ keys
REPLACE OLD LINE WITH NEW	Enter key
EXIT FROM EDIT MODE	Break key

COPY (from line#)-(to line#) (new location line)

The copy function allows portion of the current text buffer to be copied to another portion of the file. The lines included in the specifications 'from'and 'to' are copied to the new location line following the destination line. The portion of the file copied is left intact and the file is resequenced upon completion of the copy. Be aware of this if you are editing a BASIC program file because the line numbers will change and all GOTO's, GOSUB's, etc line numbers will be left intact.

Example: COPY 1100-1345 100

This would place a copy of the lines from 1100 thru 1345 following line 100.

MOVE (from line#)-(to line#) (new location line#)

The move function works almost exactly the same as the 'COPY' function only the original lines 'from-to' are removed from the file after they are copied to the new location. The file is resequenced the same in the copy function.

Example: MOVE 1100-1345 100

This would move the lines from 1100 thru 1345 to the next line following line 100.

AUTO [1 digit increment value] [line #]

Causes the computer to type sequential line numbers incremented by the specified 1 digit value. If not specified the line # will be incremented by 10. Also an optional starting line # can be specified. This is used for entering sequential text lines without having to specify line numbers, they will automatically be typed after each line is entered.

Example: AUTO 100

Enter auto line typing beginning with line '100' with a default increment value of '10'.

SIZE <Enter>

Returns the memory size that is in use and the memory that is still available. The first number displayed is the amount of memory in use or the size of the text file, the second is the amount of remaining memory.

PRINTER [command line]

Specifies that the next output operation will be output to the printer. Another command may follow the PRINTER command for ease of use.

Example: PRINTER NLINE LIST <ENTER>

This would tell the editor to list the file to the printer with no line numbers.

EXIT (Enter)

Causes control to return to 'BASIC'

NEW (Enter)

Causes the memory file buffer to be cleared and all pointers reset to the cold start condition.

LF (Enter) Allow line feed character output

This function is for those users having printers that do not automatically line feed upon receipt of a carriage return character. Normally line feed character output is inhibited, once this command is entered they will be output for each line and cannot be inhibited once enabled.

ASMB ⟨(Enter)⟩

This command causes the assembler portion of CORES-64 to be entered. Upon entry to the assembler a Pass message will be displayed, to exit from the assembler back to the Editor, enter an "E" any time the Pass message is waiting for input.

RUN ⟨Enter⟩

The RUN command has been provided so that programs which have been assembled with the object code output going to memory, can be tested without leaving CORES-64. The command will use the most recent execution address stored by the assembler, this address is listed next to the END statement on the listing. All programs to be tested with this command should exit or end by the use of a jump command to warmstart JSR $703 to insure re-entry to CORES-64 without problems. When the program is entered from this command the User Stack =$0600 & Direct Page =$06.

Note that if this command is called prior to the successful assembly of a program to memory, unpredictable results will occur.

Any time the printer is requested for an operation the status of the printer is checked for ready. If the printer is found to be in a "'NOT'" READY CONDITION", A message to that effect will be displayed and the program will wait for any key on the keyboard to be pressed, except the "BREAK" key. IF the "BREAK" key is depressed the printer output will be aborted. This will allow those users not having a printer to abort an accidental printer request and not hang up the system.

CER-COMP CORES-64 ASSEMBLER OPERATION

The CORES-64 assembler will assemble 6809 assembler source code and generate executable binary object code either to tape or memory. It will also cross assemble standard 6800 assembler source code to 6809 code compatible object code. These instructions assume that the user is familiar with assembly language programming and, in particular, the language of the M6809 Microprocessor.

Source Code:

CORES-64 will accept basically two different formats of input source code to the assembler. One form is the same as that of the EDITOR source file in memory or the same file with the line numbers removed by the remove line number editor commands (RLINES). This can be useful if the text buffer is almost completely full and space for the assembler symbol table is needed or you want to store the object code in memory. The source line format is that of the standard Motorola Assembler which is described in several manuals published by them and available from several sources.

Arithmetic Operators and Number Bases

The following operations are permitted during assembly time, which means that an expression is evaluated during the assembly and thus becomes part of the program being assembled. Numbers may be expressed in one of the three bases, which is specified by a special character for Hex and Binary. The default is decimal.

+	Addition
-	Subtraction
*	Multiplication
/	Division
$	Hexadecimal, numbers 0-9 & A-F may follow
%	Binary numbers 0 or 1 may follow

All operations are evaluated from left to right in the order in which they appear. All operations will be converted to 16 bits and truncated to 8 bits for required instructions.

Register Specification:

The 6809 has 9 registers that are accessible to the programmer, four of which are 8-bits and the other five are 16-bits. They are referenced in this assembler by the following notation:

(8)	A	Accumulator A
(8)	B	Accumulator B
(8)	CC	Condition Code Register
(8)	DP	Direct Page Register

(16)	X	'X' index register
(16)	Y	'Y' index register
(16)	U	User Stack pointer
(16)	S	System Stack pointer
(16)	PC,PCR	Program Counter & Program Counter RELATIVE

The Program counter (PC) can be used to instruct the assembler to assemble code in a Position Independent manner when used in the Indexed mode. When the Program Counter is referred to as 'PCR' it instructs the assembler to determine the offset from the current PC to some absolute address, thus making the code executable anywhere in memory.

EXAMPLE: LEAX MSG1,PCR

This would determine the difference between the current PC and the absolute address of MSG1 and use it as the offset for the PC register to calculate the effective address.

EXAMPLE: LEAX MSG1,PC

This would use the absolute address of MSG1 to add to the PC register and use that as the effective address to be loaded in the X-reg. This code is not position independent.

Assember Directives

Beside the standard machine language mnemonics CORES-64 supports several Directives. They are instructions for the assembler only and most of them do not assemble into code. The same format applies to these directives as the normal op codes. Brief explanations are given for the directives supported by CORES-64.

ORG	define new origin (PC=)
END	signal the end of the source file
RMB	reserve memory bytes
FDB	form double byte
FCC	form constant character
FCB	form constant byte
EQU	assign value to symbol
PAG	skip to top of next listing page
SPC	skip specified number of lines
NAM	specify program name (must be first line of program
OPT	set or reset assembler options
ATH	define author line contents, printed at bottom of page

ORG 〈expression,value〉

The ORG directive causes a new origin address to be used for the code which follows the directive (PC = address). The value may be a number or a label that has been previously referenced in the source file. It cannot be a reference to a label that is later defined in the program. A generated tape file will handle multiple origins.

END ⟨execution address⟩

The END directive tells the assembler that the end of the source input file has occurred. The END directive also allows for the assigment of a starting execution address for the binary tape file if created. The execution address will default to the first byte of code produced by the assembler if not specified. A label can be used for the execution address if previously defined in the program.

RMB ⟨expression,value⟩

This directive causes the assembler to reserve memory for variable or data storage. No code is produced only the address counter is changed and will be supported within a generated tape file.

FCB ⟨value,value,value,etc.⟩

This directive causes an expression to be evaluated and the resulting least significant 8 bits is stored in memory or generated within the object file. Multiple values may be used to generate several bytes of data each one separated by a comma.

FDB ⟨value,value,value,etc.⟩

This directive is essentially the same as the FCB directive only the expression is evaluated to 16 bits of data or 2 bytes for each expression.

FCC ⟨delimiter, text string, same delimiter⟩

This directive is used to create strings of characters in the object file for messages or lookup tables etc. Each character in the text string uses one byte of memory space. The two allowable formats are: a count followed by a text string in which case if the string is less than the count specified it is filled with spaces. The second form is where a text string is used by enclosing it between two characters (delimiters) that are the same character.

label **EQU** ⟨expression⟩

This directive is used to equate a symbol or label to an expression or value, no code is generated. A label must be used and an expression or value must follow the directive.

NAM ⟨file name & comment⟩

This directive is used to assign a title to the assembler listing and is also used for the tape file name. It must be the first line of any program and can only be used once in the program file.

SPC ⟨value⟩

This directive is used to tell the assembler to space down the specified number of lines in the output listing

PAG

This directive tells the assembler to skip to the top of the next page in the output listing. If the "NOPAG" option has been set the directive will be ignored.

ATH ⟨**name,comment**⟩

This directive will allow the author or a comment line to be printed at the bottom of each page in the output listing. Any text string following the directive up to 50 characters can be used for the Author line.

OPT ⟨**specifiers**⟩

The OPT directive determines how and if an object code file is to be generated. There are two options for generation object code, either one or both may be specified, they are:

M - Memory output
O - Object tape output
G - Generate data for FCC,FCB and FDB (default)
S - List symbol table after listing
P - List assembled data in page format (default)
L - List assembled data (default)

These options can also be reset by the use of the "NO" option, that is to reset the memory option you would simply use the "OPT NOM". For a .tape file to be closed correctly the option must be set at the END directive.

CORES-64 Error Messages

Error codes are used to flag source statements that are in violation of the rules and restrictions of this assembler. Error messages are output with three asterisks and the word "ERROR" followed by the error message. The line listed under the error message is the line in error. The Assembler error codes are listed below:

NAM used twice in the same program

EQU directive requires a label

Source statement syntax error

Invalid label (syntax error)

Symbol has been previously defined

Invalid op code or assembler directive

Short relative branch out of range

Address mode not allowed with op code

Byte overflow. Single byte expression converts to >255

Undefined symbol (not in table)

Invalid register for indexed operation

Re-defined symbol (Pass 2 value differs from Pass 1, usually caused by label address being referenced before assignment)

Directive operand invalid

Symbol table overflow (OUT OF MEMORY)

Assembler Overwrite (assembly to memory could destroy assembler)

ASSEMBLING WITH CORES-64

When the source file has been created or loaded by the editor and is ready to be asembled the assembler must be entered by the use of the 'ASMB' command in the editor. The assembler will display a Pass message and waits for an input, at this time a printer can be specified to direct error messages (1 pass) or the listing to the printer. The format is 'P' for printer, followed by the normal Pass selection and option. Note that object code will be generated only if the "2" or "3" pass options have been selected. The "1P" pass will clear any previous symbol table and read the source file creating a new symbol table for those file(s). The "1S" pass will keep the present table and add any new symbols to it generated by those files. The "2" pass options require that either the "1" or "3" pass has already been run if any forward references are made in the program, otherwise errors will occur. The "2" pass can be useful to generate a program listing only (2L) or object output only (2T), the "2P" pass will generate both if the output and listing have not been inhibited thru the "OPT" directive. The "3" pass options are the same as the "2" pass options only the 1 pass is automatically run and then the 2 pass is run with the specified option. This is useful for programs that have already been debugged and are error free.

Pass Options:

1P	-	Build new symbol table
1S	-	Add to symbol table
2/3P	-	Generate object and listing for specified options
2/3T	-	Generate object only for specified options (OPT)
2/3L	-	Generate listing only
2/3O	-	Generate tape object file
2/3M	-	Generate memory object file

Example:

```
ENTER PASS: 1 (P,S); 2/3(T,P,L,M,O)
>P3L
```

The printer output was selected by the first character 'P' and the "3L" pass was selected for listing only.

DIFFERENCES IN CORES-64

CORES-64 has some differences from the format standard of the 6809 auto decrement. The normal specification is ,-R and ,--R but in this assembler is can also be , R- and ,R-- which is the same as the auto increment format of ,R+ and , R++.

Because of the way CORES-64 translates 6800 to 6809 code the same code can be generated with different instructions, this can be very convenient in many cases, especially to those who have written a large amount of 6800 assembler code. Some examples are listed below:

6809	=	6800
LEAX,1,X		INX
LEAX -1,X		DEX
ANDCC #FE		CLC
ORCC#1		SEC
TFR A,B		TAB (CONDITION CODE NOT AFFECTED)
TFR B,A		TBA (CONDITION CODE NOT AFFECTED)
LDA#22		LDAA#22 (AVOIDS SYNTAX ERRORS)
PSHS B		
ADDA , S+		ABA

As you can see it can be helpful in some instances, also note that in the translation of the 6800 op codes TAB and TBA the condition codes are not affected on the 6809 as they were in the 6800. So if a program is translated which uses conditional branching following either of these instructions a "TST" instruction must be added following it to ensure that the proper condition codes are set for the branch instruction.

CORES-64 supports the forcing of direct or extended addressing by the use of the symbols " "<" and ">". But is does not support the 'DPSET' directive which is used in some assemblers to suppress error reporting when using non-zero based Direct page values. We suggest that if you require this function that the '<' be used to force direct addressing modes to be generated. Also the listing output will be flagged in the left margin next to the address with a '*' when non-zero direct page addressing is forced. The listing is also flagged with a '>' in the left margin, when an Extended Branch or Jump is not necessary. Also CORES9 does not support the instruction "LSL" as it is logically and machine code equal to the "ASL" instruction.

```
0010  NAM DEMO THIS IS A DEMONSTRATION PROGRAM
0020  ORG $3300 BEGIN ABOVE PROGRAM
0030  OPT NOG
0040  ATH BILL VERGONA
0050  SPC2
0060   * COMMENT LINE IS NOTED BY AN ASTERISK '*' IN COLUMN 1
0070   * THIS IS A CO-RES69 EDITOR/ASSEMBLER DEMONSTRATION PROGRAM
0080   *
0090  SPC2
0100   START PSHS DP,U SAVE CO-RES9 REGS
0110  CLRA SET DP FOR BASIC
0120  TFR A,DP
0130  LEAX PROMPT,PCR POINT TO INPUT MESSAGE
0140  BSR DISPLY DISPLAY MESSAGE
0150  LEAX INBUFF,PCR POINT TO INPUT BUFFER
0160   INLOOP JSR $B538 INPUT CHAR/FLASH CURSOR
0170  TSTA CHECK NO KEY
0180  BEQ INLOOP WAIT FOR KEY
0190  JSR [$A002] ECHO CHAR TO SCREEN
0200  CMPA#$D RETURN KEY
0210  BEQ INEND GO IF END
0220  STA , X+ PUT CHAR IN BUFFER
0230  BRA INLOOP GET NEXT
0240   INEND LDA#4 END OF LINE MARK
0250  STA, X
0260  LEAX INBUFF,PCR POINT
0270  BSR DISPLY DISPLAY BUFFER INPUT
0280  PULS DP,U,PC RESTORE REGS & RETURN
0290   *
0300   *  NOTICE (  ) MAY ALSO BE USED FOR INDIRECT ADDRESSING
0310   *
0320   OUTCHAR JSR [$A002] DISPLAY CHAR
0330   DISPLY LDA, X+ GET CHAR
0340  CMPA#4 END CHAR
0350  BNE OSUTCHR OUTPUT IF NOT
0360  RTS RETURN
0370   PROMPT FCB  $D
0380  FCC /ENTER A LINE OF TEXT ENDING WITH THE 'ENTER' KEY/
0390  FCB $D,$A,4
0400   INBUFF EQU * INPUT BUFFER BEGINS HERE
0410  END
```

```
---- PAGE 001    DEMO  THIS IS A DEMONSTRATION PROGRAM

00010                          NAM    DEMO  THIS IS A DEMONSTRATION PROGRAM
00020 3300                     ORG    $3300 BEGIN ABOVE PROGRAM
00030                          OPT    NOG
00040                          ATH    BILL  VERGONA

00060       * COMMENT LINE IS NOTED BY AN ASTERISK '*' IN COLUMN 1
00070       * THIS IS A CO-RES69 EDITOR/ASSEMBLER DEMONSTRATION PROGRAM
00080       *


00100 3300  34     48    START  PSHS   DP,U   SAVE CO-RES69 REGS
00110 3302  4F                  CLRA          SET DP FOR BASIC
00120 3303  IF     8B           TFR    A,DP
00130 3305  30     8D 002F      LEAX   PROMPT,PCR POINT TO INPUT MESSAGE
00140 3309  8D     26           BSR   DISPLAY DISPLAY MESSAGE
00150 330B  30     8D 005D      LEAX   INBUFF,PCR POINT TO INPUT BUFFER
00160 330F  BD      B538 INLOOP JSR    $B538   INPUT CHAR/FLASH CURSOR
00170 3312  4D                  TSTA          CHECK NO KEY
00180 3313  27     FA           BEQ    INLOOP  WAIT FOR KEY
00190 3315  AD     9F A002      JSR    [$A002] ECHO CHAR TO SCREEN
00200 3319  81     0D           CMPA    $D     RETURN KEY
00210 331B  27     04           BEQ    INEND   GO IF END
00220 331D  A7     80           STA    ,X+     PUT CHAR IN BUFFER
00230 331F  20     EE           BRA    INLOOP  GET NEXT
00240 3321  86     04     INEND LDA    #4      END OF LINE MARK
00250 3323  A7     84           STA    ,X
00260 3325  30     8D 0043      LEAX   INBUFF,PCR    POINT
00270 3329  8D     06           BSR    DISPLY  DISPLAY BUFFER INPUT
00280 332B  35     C8           PULS   DP,U,PC RESTORE REGS & RETURN
00290       *
00300       * NOTICE "( )" MAY ALSO BE USED FOR INDIRECT ADDRESSING
00310       *
00320 332D  AD     9F   A002    OUTCHR JSR    [$A002] DISPLAY CHAR
00330 3331  A6     80    DISPLY LDA    ,X+     GET CHAR
00340 3333  81     04           CMPA   #4      END CHAR
00350 3335  26     F6           BNE    OUTCHR  OUTPUT IF NOT
00360 3337  39                  RTS           RETURN
00370 3338  0D           PROMPT FCB    $D
00380 3339  45                  FCC    /ENTER A LINE OF TEXT ENDING WITH THE 'EN

00390 3369  0D                  FCB    $D,$A,4
00400       336C         INBUFF EQU    *       INPUT BUFFER BEGINS HERE
00410       3300                END

TOTAL ERRORS 00000
```

****
DEBUG MODULE COMMANDS
****

The DEBUG module is an extension of the Editor/Assembler package. It allows the user to test and debug machine language programs that have been assembled to memory. To enter into the DEBUG module from the EDITOR enter the command "DBug" and hit the enter key. The DEBUG module sign on message should be displayed and a ")" character will be displayed for a command prompt. This is the only way the DEBUG module can be entered, it cannot be entered directly from the ASSEMBLER. Once in the DEBUG module only its commands may be used. To return to the EDITOR/ASSEMBLER use the "COres" command followed by the enter key. The "READY" prompt will be displayed when you return to the EDITOR.

The DEBUG module commands are similar to the Editor commands in that they can be abbreviated by the first two characters of the command. The command input line is buffered and will recognise the backspace, clear, break and enter keys for easy error free command entry. Each command line is terminated by hitting the "ENTER" key. Output from any command may be temporarily paused by hitting any key and resumed upon hitting another key.

DEBUG MODULE Commands:

ME	<address>	Memory examine & change
SB	<address> <etc>	Set and/or display breakpoint
RB	<address>	Remove one or all breakpoints
RS	<value> <name>	Set and/or display registers
GO	<address>	Goto address with stack
DM	<begin> <end>	Dump Memory in Hex & ASCII
FM	<begin> <end> <byte>	Fill Memory with data byte
FI	<begin> <end> <byte> <etc>	Find Memory byte sequence
BM	<begin> <end> <destination>	Move block of memory
DA	<begin> <end>	Disassemble memory file
CO		Exit monitor back to CORES64
IZ		Re-Initialize DEBUG

DEBUG error codes

AD - Address error begin> end
CD - Command error
CE - Conversion error on address or data byte

MEmory <address> Memory examine & change

This function allows the user to examine and change the contents of a specific memory location on a byte by byte basis.

When the function is called and a valid hex address has been entered it will display both the address & data contained in that location of memory in hex. If the address was not a valid hex address or none was entered the last address stored in BEGIN1 will be used." Once the address & data are displayed the user can change that byte, and/or move forward or move backward thru memory. If the data is to be changed simply enter the new hex 2-digit value, if for some reason the new value cannot be stored correctly a '?' will be displayed and the next location will be displayed normally. If an up arrow '^'is entered the previous location will be displayed and if a carriage return 'cr' is entered the function is ended. Any other non-hex character will cause the next location to be displayed.

Example:

```
>ME 3FFE<cr>
3FFE 49                 period advances to next location
3FFF 98 55              hex value changed to 55
4000 27 11?             new value not changed correctly
4001 31 ^               display previous location
4000 27<enter>          end function
```

SBreak <address> <etc> Set and/or display Breakpoints

The BReakpoint function allows the user to set program breakpoints in memory in order to de-bug programs. If no valid address is entered the function simply displays the contents of the breakpoint table. If a valid address was entered and the table is not full a breakpoint (SWI) will be set in memory and the entry set in the table. It then displays all breakpoints set in the table. When a breakpoint is executed in a program and the SWI interrupt jump vector in memory has not been changed a dump of the registers will be displayed on the system console and the original code will be restored in memory removing the breakpoint. Several breakpoint addresses may be entered on one command line.

Example:

```
)SB 1000 13FF 1103          Set SWI at 1000, 13FF & 1103
1476 1000 13FF 1103         Breakpoint table contents
    shows 1476 was previously set & the new ones set
```

RBREAK ⟨address⟩ Remove one or all Breakpoints

This function allows breakpoints previously set by the SB command to be removed individually or all at once. If a valid address was entered and it is found in the break table only that breakpoint will be removed. If no address is entered all breakpoints in the table will be removed. Notice that "RESET" will not clear the breaktable unless a system initialisation is required and that breakpoints encountered in a program are automatically removed if they are contained in the breaktable. Only one breakpoint address can be removed at a time.

Example:

```
>RB 13FF          remove breakpoint at address 13FF
>RB               remove all breakpoints from table
```

RSet ⟨value⟩ ⟨name⟩ Set and or display register contents

This function allows the user to change the value contained in a particular processor register on the defined stack. If no value was entered the function simply displays all the system registers and their contents. If a valid hex value and register name were entered the contents of that register will be replaced by the value entered. The registers will then be displayed for visual verification of the change.

Register Names:

C - Condition code	A - Accumulator A
B - Accumulator B	D - Direct Page Register
X - Index register X	Y - Index register Y
U - User Stack pointer	S - System Stack pointer
P - Program Counter	

Example:

```
)RS 99 A          change the contents of Acc-A to 99
)RS 100 X         change the contents of IX to 0100
)RS               display register contents
```

GO ⟨address⟩ Goto defined address with stack

This function allows the user to resume processing of a program that was interruped by a breakpoint or other interrupt that caused a system trap entry. If a valid address was entered with the command that address will be placed in the program counter register on the stack prior to calling an "RTI" return from interrupt.

Example

```
>GO ⟨cr⟩          resume program at address contained in stack PC
>GO 1000 ⟨cr⟩     begin execution of program at address $1000
```

DMemory ⟨begin⟩ ⟨end⟩ Dump Memory in Hex & ASCII format

The Dump function allows the user to display & examine areas of memory much easier than using the memory examine & change function. The output is formatted with 8 bytes of data per line with the ASCII characters underneath if printable. The contents of memory between the begin and end addresses will be displayed, if either the begin or end address is omitted or invalid the function will be aborted and an error displayed. If the output is directed to the printer the format will be 16 bytes per line followed by the ASCII characters on the same line.

Example:

```
>DM 100 10F                    Display memory from $0100 thru $010F
0100 16 00 79 7E 02 4E 44 55
---- .  .  v ~  .  N  D  U

>?DM 100 10F                   Display $0100 thru $010F on printer
```

FMemory ⟨begin⟩ ⟨end⟩ ⟨byte⟩ fill memory with data byte

This function allows the user to fill a defined segment of memory with a specific data pattern. All three parameters must be entered with the command or an error will be reported. This function can be useful for initializing memory for a program or filling memory with a SWI (3F) for trying to trap runaway programs.

Example:

```
>FM 400 600 3F                 fill memory from 400 thru 600 with 3F
>FM 2000 4000 00               clear memory from 2000 thru 4000
```

FInd ⟨begin⟩ ⟨end⟩ ⟨byte⟩ ⟨etc⟩ Find byte sequence

This function will allow the user to search a defined segment of memory for a predefined byte sequence. Any number of bytes can be searched for 1,2,3,4,5 etc. depending upon how many are entered. At least the begin, end, and 1 byte to search for must be entered or an error will be reported. If the specified string of bytes is found in the range of memory specified the address and data byte of the previous location, search bytes, and the one data byte following the string will be displayed. The search will then continue until the end address is reached, displaying the information for each occurance of the byte sequence.

Example:

```
>FI A000 BFFF A3 90
A746 BD A3 90 5A
AA68 7E A3 90 9F
```

BMove ⟨**begin**⟩ ⟨**end**⟩ ⟨**destination**⟩ Block memory move

This function will move a defined block of memory from one place in memory to another. The begin and end addresses define the block of memory to be moved and the destination address is where it is to be moved to. If any of the parameters are not entered an error will be displayed.

Example:

```
>BM 1000 1500 6000          Move 1000 thru 1500 to 6000
```

DAsmb ⟨**begin**⟩ ⟨**end**⟩ Disassemble memory into assembler format

This function will dis-assemble a specified segment of memory displaying it in an assembler or code format. It will display the address of each instruction, op code, and operand byte(s). All relative branch instruction will also display a '>' followed by the destination address of the branch instruction. This function is not fool proof by any means and some sequences of memory will be decoded as instructions which are really text characters or data bytes. It is only designed to be an aid in debuging and disassembling programs.

Example:

```
>DA A00E A06F                Disassemble from A00E thru A06F

A00E 10CE 03 D7
A012 86   37
A014 B7   FF23
A017 96   71
A019 81   55
A01B 26   52       >A06E     branch destination address
A01D 9E   72
```

COres Exit the Debug module back to CORES

This function simply allows the user to exit from the Debug module back into Basic. Once the monitor has been exited in this manner the RESET switch will no longer return to the monitor but to Basic.

IZ Re-initialize monitor

This function simply re-initializes the monitor to a cold start condition. Prior to initializing all previously set breakpoints will be restored. This can be useful if some portion of the monitor temporary storage were modified or for any other reason the monitor may not be functioning correctly.

? (Printer Switch)

This function redirects the screen output to the printer

C omprehensive Instruction Manual

O utput Assembled Machine Code to Memory, Printer or Binary Tape File

R esident Monitor Programme for Debugging

E xtensive Editing Commands

S upports Multiple Origins and RMB.

Produces ASCII Tape Files which can be loaded under BASIC

Automatic Memory Sizing for 32K or 64K machines

In our opinion the Best Tape Based Editor/Assembler yet for the Dragon

MICRODEAL