THE MICRO WORKS

SOFTWARE DEVELOPMENT SYSTEM SDS 80C OWNER'S MANUAL

P.O. BOX 1110 DEL MAR, CA 92014 714-942-2400

THE MICRO WORKS

P.O. BOX 1110 DEL MAR, CA. 92014 [714] 942-2400
UPS ADDRESS: 1942 S. EL CAMINO REAL, ENCINITAS, CA 92024

SDS80C

SOFTWARE DEVELOPMENT SYSTEM
EDITOR / ASSEMBLER / MONITOR

 Version II.0

WARNING: Opening this ROMPACK will void your warranty!

MICRO WORKS SOFTWARE DEVELOPMENT SYSTEM

SDS80C

for the Radio Shack COLOR COMPUTER

WARNING: DO NOT OPEN the SDS80C Rompack.
See the last page of this manual for warranty information.

TABLE OF CONTENTS

Written By
Andrew E. Phelps
The Micro Works, Inc.
P. O. Box 1110
Del Mar, California 92014

SECTION 1 - INTRODUCTION TO SDS80C

***** GETTING STARTED

Turn off the computer, plug in the SDS80C cartridge, and turn on the computer. WARNING: Never plug or unplug a cartridge with power on!

DO NOT OPEN the SDS80C Rompack. This voids the warranty. See the last page for warranty information and the update registration form.

Do not touch the exposed edge connector on the Rompack. In conditions of high levels of static electricity this could damage the Rompack.

The editor will come up with a screen which is blank except for the top line. The number on the right side of the top line indicates the number of bytes of memory which is available for a text buffer. The other number is there to indicate your position within a file; since it is currently zero it displays a row of asterisks.

The editor commands are described in the next chapter. Briefly, however, you can:

Use the arrow keys to scroll throughout the text buffer;
Type "L" to get into line insert mode; Break exits the mode;
Type "D" to get into line delete mode; Enter exits the mode;
Type "X" to get into text exchange mode; Enter exits the mode;
Type any of the 15 other commands.

There are three distinct programs in SDS80C. The first one is the editor; this is run when SDS80C is first started up. The second is the assembler; this is run by typing "@" from the editor and will assemble whatever is in the editor's buffer. The third is the monitor ("ABUG"); this is a special version of the powerful Micro Works CBUG monitor, and has been adapted for use in overseeing the execution of programs produced by the assembler. The "*" command in ABUG returns control to the editor.

***** THE USES OF SDS80C

The Software Development System (SDS80C) is intended to aid in the development of assembly language programs on the Color Computer. It contains a screen-oriented text editor which is optimized for use with assembly language source text. The assembler has been designed for compatibility with standard 6809 source format, so that any 6809 code may be entered without modification.

The text editor may be used with any text, and can edit Basic programs or documents where the line length is limited to the width of the screen. It is not designed for this, however, and its use in editing text other than assembly language source is not supported.

***** PROGRAM DEVELOPMENT

The following is the usual sequence followed in program development.

A. Run SDS80C.
B. Load the most recently saved source tape.
C. Edit the source. Be sure to update the date and revision level comments; it is easy to lose track of which version is which.
D. Periodically run the assembler to check syntax.
E. If substantial changes have been made, write a new tape. Alternate between at least two tapes. Try using tally marks to keep track of which tape is most recent.
F. To verify a tape without losing source, try appending it.
G. Assemble to memory and try running it.
H. When a program crashes, reload the last source tape.
I. Add SWI instructions into the source for pause-and-dump during debugging. End your program with SWI/LBRA START.
J. If object won't fit into memory, write to tape and then load the tape; reload the source afterwards.
K. When the program is sufficiently debugged, write an object tape so that it may be loaded without loading the source. Remember to remove all SWIs.

***** SAMPLE PROGRAM

To get started, try entering this sample program. (Refer to the next section for instructions on using the editor.)

```
        NAM INVERT
START   LDX #$400 SCREEN ADDRESS
LOOP    LDA 0,X GET CHARACTER
        EORA #$40 INVERT COLOR
        STA ,X+ SAVE; INCREMENT X
        CMPX #$600 END OF SCREEN?
        BLO LOOP DO WHOLE SCREEN
        SWI CALL ABUG
        BRA START AND DO IT AGAIN
        END START
```

When it is entered correctly, type "@M" (ENTER) to assemble it. Type "G" to run it; you may type "G" any number of times because of that last branch statement. Finally, type "*" to return to the editor from ABUG.

SECTION 2 - THE EDITOR

***** INTRODUCTION

The editor is screen-oriented; what is seen on the screen is what is in the text buffer. It is impossible to have any further information in the file than what is visible. The cursor may be moved throughout the file using the arrow keys. The size of the source being edited is limited to the available memory of the computer.

While in the editor, all keys will automatically repeat while held down. For example, holding down the down arrow key will scroll through the entire source file. This feature is deactivated in the assembler and in ABUG.

When in the "command level", the top line will display two numbers. The rightmost number shows the number of available bytes of memory and to the left is the number of bytes down from the top of the screen at the current position. If either number equals zero a line of asterisks will appear instead; this is the normal condition for the left number when the screen is at the top of the file. If the right number becomes zero then the buffer is full; some text should be deleted in order to continue editing.

***** COMMAND MODE

In command level, a single key is pressed to cause a specific action. The meaning of the keys is as follows:

ENTER - go down to next line
SPACE - move cursor to next character, skipping blocks of spaces
(left arrow) - move cursor left one character
(right arrow) - move cursor right one character
(up or down arrows) - move cursor up or down one line
(shift left arrow) - move cursor to left edge
(shift right arrow) - move cursor to right of last text on line
L - insert lines
D - delete lines
X - exchange text; insert or delete characters
F - find string (-F searches backwards)
C - find and change string (-C searches forward from the beginning)
A - do last find or change again (for globals)
P - go forward one page (-P goes backwards one page)
T - copy block of text
M - move block of text
J - jump to beginning or end
W - write file to tape
R - read file from tape
@ - assemble file
& - recover file after reset

***** CONTROL CHARACTERS

Certain keys, such as ENTER or BREAK, are considered control keys. In the command level, functions are assigned to keys (and to shift-keys) without regard to whether or not they are control characters. When entering parameters and data to particular commands, however, it is useful to keep in mind the set of control characters versus data characters.

The following are always control characters: ENTER, BREAK, left arrow, right arrow, down arrow, (NOT up arrow), shift left arrow. In any mode in which the function of one of these keys is not defined, it has no effect.

Shift zero is the lowercase toggle and functions in the same way as in BASIC. It may be pressed at any time regardless of mode. An "L" will appear in command mode on the top line when in lowercase mode.

Shift-@ (the pause function in BASIC) is not used by the editor and may in some modes be treated as a space. The CLEAR key is not used and is ignored.

The up arrow key is not a control key and generates a printable up arrow symbol. Shift up arrow generates a printable left arrow. Shift down arrow generates a "[" while shift right arrow is "]". These characters are all normal text and are allowed wherever text is entered.

In the "enter string" mode only, down arrow and right arrow are treated as text characters. They display as inverted "[" and inverted "]". In that section of this document they are shown as "[" and "]" but it should be remembered that they are distinct from the non-inverted variety of these characters.

***** ENTERING STRING PARAMETERS

Some of the above commands require the entry of one or more strings. These are:

- F - enter the string to find.
- C - enter the string to change, and another to change it to.
- W - enter file name for tape header.
- R - enter file name to search for (or just ENTER).
- @ - enter assembler options.

When a string is required, the top line will be blanked and the cursor placed at the beginning of it. Ordinarily the top line is reserved for prompts and this is the only time the cursor will be allowed there. When entering a string, the following characters have meaning:

ENTER - the string is done; the position of the cursor implies the length of the string.
shift left arrow - clear string and start again.

left arrow - back space.
break - abort command.
any other key - text of string.

When two strings are required, such as in "Change", they are each terminated by "ENTER".

***** THE JUMP COMMAND

Pressing "J" in command mode will enter JUMP mode. At this point you may enter:

B - jump to beginning.
E - jump to end.
F - jump to beginning and go to FIND. (JF may be thought of as "find first occurrence of ...").

***** THE EXCHANGE TEXT COMMAND

Pressing "X" in command mode will enter the "XCHANGE" mode; this is used to write data directly onto the screen and to move characters left and right. It is the primary method of correcting lines which have already been entered. You may enter:

ENTER - leave exchange mode.
BREAK - restore file and leave.
text - enter text and advance cursor.
left arrow - backspace cursor.
shift left arrow - delete characters and close up line.
right arrow - move characters to make room in line.

***** THE DELETE LINES COMMAND

Pressing "D" in command mode will enter DELETE mode. At this point you may enter:

ENTER - leave delete mode.
down arrow - delete one line.
right arrow - delete 32 characters. This is useful for deleting a line while keeping its label.

***** THE INSERT LINES COMMAND

Pressing "L" in command mode will enter the "INSERT LINES" mode. This is the command used for entering the bulk of a text file. Note that BREAK exits the command; there is no key for restore file. You may type the following:

ENTER - go to next line.
text - insert text and move cursor.
SPACE - in the first column, this causes a tab to mnemonic field.
left arrow - backspace and erase last character.
right arrow - insert space and move cursor.
BREAK - return to command level.
shift left arrow - delete line.

***** FIND STRING

To find the next occurrence of a string, type "F". The top line will go blank and you may enter a string to search for. When ENTER is pressed, the cursor will move to the next line containing that string. The search is started at the first line after the line containing the cursor.

To find each subsequent occurrence of that string, press "A" (for "again"). When no more occurrences may be found the cursor will move to the end of the file.

To find the first previous occurrence of a string, type "-F" instead of "F". Then each "A" will search backwards until the top of the file is reached.

The string which is typed will only be matched if it is bounded by non-alphanumeric characters. For example, if all occurrences of the variable name "DA" are to be found, the editor will not match with strings such as "LDA" or "DAA" since these are bounded by an alpha-numeric character. If you wish to find all occurrences, including those which are bounded by alphanumerics, enter the string beginning with a quote ("). For example, a command to find "DA will match any line containing a DA no matter what its context. (Note that only the begin quote, not an end quote, is entered.)

All trailing spaces on each line are removed in the text buffer. All leading spaces are removed save one which is left to serve as a tab character. This format can affect the Find command. Find XYZ(space) will not find XYZ if it is the last thing on the line, since the trailing space is not there in the text buffer. Find (space)LDA will work, but Find (space)(space)LDA will not work since only one leading space is retained.

There are two special characters which the string may contain. Down-arrow (which on the screen displays as an inverted "[") will let the string match only if there is a new line in the buffer. Find "[XYZ will only find XYZ if it is a label, and find "XYZ[will only find XYZ if it is the last thing on the line. Right-arrow (which displays as an inverted "]") will match with any character as long as it is on the same line. Find "]XYZ will find XYZ as long as it is not in the label field; and find X]Z will find XAZ, XBZ, etc. Note that using "[" and "]" at the ends of strings generally requires that the string start with a quote.

Find "[XYZ will not find XYZ if it is in the first line of the file, since there is no carriage return preceding it to match the "[".

***** CHANGE STRINGS

The "C" command is used to find the next occurrence of a string and to change it to another string. Although it can be used for a single instance of the string, the "X" command is better at that. The main use is along with the "A" command to effect a global change; the "A" key may be held down to rapidly change every occurrence in the buffer.

The operation of "C" is similar to "F", and the "F" command should be understood first. After typing "C" or "-C", two strings are entered, each terminated with a carriage return. The first string is then found under all of the same rules as with the "F" command. The second string replaces the first string.

If the first string contains down- or right-arrows (inverted "[" or "]"), the characters they match are replaced along with the rest of the string. Therefore care must be used with these characters. Changing "[XYZ to PDQ will not only change XYZ to PDQ but will also have the effect of appending it to the line above, as the line separator character will be removed. If the second string contains a down-arrow character, the line will be divided into two lines.

***** MOVE BLOCK OF TEXT

To move a block of text to another location within the text file, you must:

1. Position the first line to be moved at the top of the screen.
2. Position the cursor on the line below the last line to be moved if you can; if it is off the screen leave the cursor at the bottom of the screen.
3. Type M.
4. If the cursor isn't already below the last line to be moved, use shift-down arrow to get it there.
5. Use the up and down arrow keys to move the rest of the source file past the fixed area. The area from the original top of the screen through the line above the cursor will remain fixed and will not scroll.
6. Type ENTER or BREAK when the text is in the proper position.

***** COPY BLOCK OF TEXT

The Copy function is used to create two adjacent copies of a block of text; the Move function is then used to drag the new copy to wherever it is needed.

To copy a block:

1. Position the first line to be copied at the top of the screen.
2. Type T.
3. Use the down arrow key to move lines off the top of the screen.
4. Each line moved off the top will be written twice.
5. When the last line to be copied has been moved off the top, hit ENTER or BREAK to return to command mode.

***** WRITE FILE TO CASSETTE

To save the text file, type "W", then enter a string to be used as the name of the file on tape. (Only eight characters are used, longer names are truncated.) A null string (ENTER only) will work, but it is

not recommended as it makes it harder to keep a tape library straight.

There will be a pause while a header is written, then the file will be written to tape. As it is written, it will appear a block at a time on the screen in compressed format. Don't panic. After the end-of-file is written the screen will be restored to normal format.

***** LOAD FILE FROM CASSETTE

To load or append a saved text file, type "R" for Read. Then type "L" for Load or "A" for Append; the "L" will wipe out any text in the buffer at that point. Enter a string; this is the file name to search for. If no string is entered (ENTER only) the first file encountered will be loaded. An "S" will now appear on the screen and the familiar tape load procedure will occur. The screen will be reloaded when the end-of-file is encountered.

***** CALLING THE ASSEMBLER

To assemble the source file, type "@". Then enter a string which contains the parameters to the assembler; for example, LSM means list to screen, display sorted symbol table, and generate object to memory. See the assembler section for a complete decription of the parameters.

***** CALLING ABUG

ABUG is called automatically from the assembler if the object code was written to memory. To get directly from the editor to ABUG, type "@" (as if to go to the assembler) but for parameters type "=" (return). This will cause the assembler to call ABUG without doing an assembly. In ABUG, type "*" to return to the editor.

***** RECOVERY OF THE FILE

If an object program crashes, or for any reason it is necessary to reset the computer, the safest course of action is to reload the most recent source tape. It is possible, however, to try to recover the file from RAM.

Type the command "&". This will put into the buffer anything that looks like the old text buffer. There will probably be some extra garbage at the end. Jump to the beginning, find the END statement, and use the Delete command to remove anything else.

SECTION 3 - THE ASSEMBLER

***** INTRODUCTION

The Micro Works 6809 Assembler accepts source statements in standard 6809 assembly language and produces object code to tape or memory. It is designed to be co-resident with the Micro Works screen editor, and expects the source statements to be in memory in the format used by the editor. It can produce a listing of the source and object code to the screen or to a printer.

It is assumed that the user is familiar with 6809 assembly language, and only a brief description of this language is given in this manual. For those who wish further information on language syntax and programming techniques, an excellent book on the subject is available through the Micro Works. The SDS80C assembler supports all standard instructions, address modes, and mnemonics, and in addition it features the following:

- Support of all 6800 instructions for cross-assembly;
- Local labels;
- Conditional assembly.
- Pause/Break/Speed control of listing.

***** ASSEMBLER OPTIONS

When the assembler is called from the editor, a string of options is requested. These may be in any order. The available options are:

L	Produce a listing;
S	Produce a sorted symbol table;
M	Generate object code to memory;
T	Generate an object cassette tape;
!	Start listing in single step mode;
3	Send output to 32 column printer;
4	Send output to 40 column printer;
8	Send output to 80 column printer;
=	Do not assemble; go to ABUG.

If L is not specified, only error lines are listed.

If M is specified, the object code will be generated to memory starting at the address directly following the end of the source buffer. The end of the area available is determined by the stack pointer. Any ORG statement which would cause it to generate code not in this area will cause an error and suppress code generation.

Pressing the spacebar during a listing will put the assembler in single-step mode, and any other key will restart it. The option "!" will start the listing in single-step mode.

Any of the options 3, 4, or 8 will direct all output to the RS232

port. Use 3 for a 32-column printer which generates a CRLF on the 32nd printed column; use 8 if each source line will fit on one printer line. All other printers use 4. Option 8 will tab the comment field.

The option "=" overrides all other options. No assembly is done, and control is passed to the monitor ABUG. This is merely a way to call ABUG from the editor. Normally ABUG is called after assembly if the option M was specified.

The option "T" generates an object tape. The name for the file is taken from the NAM statement which should appear at the top of the program. If there is no NAM statement the file name will be all spaces. If object code is written to tape, there is this restriction: The object code must be in one contiguous block. No ORG to a previous location is allowed. A forward ORG or an RMB will generate $3F's (SWI) to fill the unused area.

***** PAUSE / BREAK LISTING CONTROL

When the listing is sent to the screen, several keys are used to slow down or stop the listing so that it can be read.

Pressing the spacebar enters single step mode. Each time it is pressed another line is displayed. Any other key will exit single step mode. Single step mode is entered automatically whenever an error message is printed. The listing may be started in single step mode with the option "!". It is possible to be in single step mode even when there is no output.

The listing may also be stopped with the shift-@ key, as with BASIC.

There is ordinarily a small time delay in the displaying of each character, in order to make it possible to read the listing as it goes by. The listing may be sped up with the "S" key to skip past areas quickly. Repeated "S"s will toggle the speed feature.

Pressing the BREAK key will abort the assembly, close any open tape output, and return control to the editor.

***** EXECUTION OF THE OBJECT CODE

After object code is generated, it is generally run under control of the ABUG monitor.

If option "M" was specified, control will transfer to ABUG after assembly is finished. The ABUG command "*" returns control to the editor; "G" will execute the code. See the chapter on ABUG for details of these and other commands.

The END pseudo-op should be used to specify a transfer address. If an expression is included as the operand field of the END statement, this address is written to the cassette tape header and sent to ABUG. If no transfer address is specified, the beginning of the generated

code is assumed.

***** ERROR MESSAGES

The following are the possible error messages:

ERROR - BYT	A one-byte value is required; the expression evaluated to a number outside the range -128 through 255.
ERROR - DDS	Doubly-defined symbol; the previous value is retained.
ERROR - FAR	A short branch is made to a symbol too far away.
ERROR - FOR	This statement requires an expression with no forward reference; a symbol used is either forward referenced or undefined.
ERROR - IMM	This statement requires an immediate-mode operand.
ERROR - LAB	Error in label field.
ERROR - MNE	Undefined mnemonic.
ERROR - OBJ	Attempt to generate object code in illegal area. See the constraints in the section below.
ERROR - OVF	Overflow of symbol table - fatal.
ERROR - SYN	Syntax error in operand field.
ERROR - TER	Terminator expected after operand. Can be caused by a syntax error in an expression.
ERROR - TRU	A label has been truncated to six letters.
ERROR - UND	Undefined symbol

***** OBJECT CODE CONSTRAINTS

Object code may be sent to memory, cassette, or both.

When object code is sent to memory, it must be to an area between the end of the source buffer and the beginning of the symbol table (which grows on the stack). If there is no ORG statement, the option M causes the program to start directly after the source buffer.

When object code is sent to tape, it must be in one contiguous block. That block will start with the first statement which generates output. An ORG to a forward location will cause the tape to be filled with $3Fs until that location is reached. A backwards ORG (one to a previous location) is not allowed.

Object code may not be generated above address $FEFF. This would be in the I/O area.

Anytime one of the constraints above is violated, the message ERROR - OBJ will be generated. If there is tape output it will cease at this point.

The use of the ORG statement is generally not required and is not recommended.

The following example shows first a program which generates a very long object tape, then one which generates a much shorter tape:

```
      NAM LONGPROG
VAR   BZS 1
TABLE RMB 1000
START  ...

      NAM SHORTER
TABLE RMB 1000
VAR   BZS 1
START  ...
```

***** EXPRESSIONS

A general expression processor is used to evaluate any portion of an operand field where an arithmetic value is needed. Thus anywhere a symbol may be used as an operand or part of an operand, one may use an expression of arbitrary complexity. Examples:

```
      LDX #3-(XYZ*$22+'&!R3)
      LDA [1-(2-(3-(4-(5-6)))),X]
      FDB 1,$2,%11,@4,5H,110B,7Q,'8-'0
```

Note that spaces are not allowed in an expression as they are used as field terminators. Commas are not allowed in expressions; they separate expressions (as in the FDB intruction above) or can signal indexed mode (as in the LDA instruction).

Parentheses may be used at will to specify the grouping of the operations. (Brackets may not be used as these are used for indirect addressing.) In the absence of parentheses, the order is (1) unary minus, (2) all operations except + and - (left to right), (3) plus and minus last. This is the standard order of evaluation which one would expect.

All operations are performed in 16 bit arithmetic and no overflow or carry conditions are recognized.

In addition to +, -, *, and /, the logical operations AND (!.), OR (!+), and EXCLUSIVE OR (!X) are included, with the same precedence as * and /. Also included are shift left, shift right, rotate left, and rotate right (!<, !>, !L, and !R respectively), which shift the left operand the number of bits specified in the right operand. Examples:

```
      5!<2   is 20
      5!>2   is 1
      5!R2   is 16385
```

Constants can be specified in base 2, 8, 10, or 16. Default is base 10. A constant may be preceded by a %, @, &, or $ to indicate bases 2, 8, 10, or 16 respectively. Alternatively a suffix of B, Q (not O), or H may be used for bases 2, 8, or 16; there is no suffix for base 10. A constant may not start with a letter or it will be interpreted as a symbol; therefore FFH must be written OFFH so that it will start with a digit. Better yet, write $FF; the prefix form is more widely

used as it is more readable.

Ascii constants are preceded by a single quote. There is no end quote.

The symbol "*" may be used to represent the value of the assembler's PC at the beginning of the line. For example, the statement FDB LABEL-* will store the relative address of LABEL.

Expression syntax is as follows (the ":" separates syntax options):

```
EXPRESSION = TERM OP1 EXPRESSION  :  TERM

OP1 = +  :  -

TERM = FACTOR OP2 TERM  :  FACTOR

OP2 = *    :          (multiplication)
      /    :          (division)
      !.   :          (logical AND)
      !+   :          (logical OR)
      !X   :          (logical XOR)
      !<   :          (shift left)
      !>   :          (shift right)
      !L   :          (rotate left)
      !R              (rotate right)

FACTOR = symbol  :
         decimal number  :
         & decimal number  :
         hex number H  :          (must start with digit 0-9)
         $ hex number  :
         octal number Q  :
         @ octal number  :
         binary number B  :
         % binary number  :
         ' character  :           (7-bit ascii constant)
         - FACTOR  :              (unary minus)
         + FACTOR  :              (unary plus)
         ( EXPRESSION )  :        (grouping with parentheses)
         *                        (value of PC at beginning of line)
```

***** SYMBOLS

Symbols may be any string of letters and numbers, starting with a letter, and up to six characters long. Symbols longer than six characters will assemble correctly but will generate a TRU error message to indicate that they were truncated.

Symbols may also contain the characters "." and "$", but not as the first character.

Symbols are sorted alphabetically as they are encountered and a sorted symbol table may be printed after the listing by specifying option S.

There are no reserved symbols; any symbol may be used. However, this is not always a good idea. The symbols A, B, and D should not be used as they cannot be used in indexed statements such as LDD A,X (register A will be assumed). The symbol X will assemble correctly in all cases, but should not be used; LDA X had best generate an ERROR - UND message since 6800 programs could use this to mean LDA ,X. For compatibility with other 6809 assemblers, it is recommended that the symbols X, Y, U, S, PC and PCR be avoided; they only lead to confusion anyway.

Symbols are defined by placing them in the label field, ie., at the beginning of a statement. They are allowed in any statement. They will be given the value of the program counter at the beginning of the statement, unless the statement is EQU which gives the symbol the value of the operand. A symbol is not considered defined until after its statement is assembled; the statement:

```
XYZ  RMB  XYZ-$100
```

will generate an error since an RMB statement requires its operand to be defined during the first pass through the source, and XYZ will not be officially defined until after the RMB is assembled.

The assembler must know during pass 1 how long to make each statement. If it is necessary to know the value of a symbol in order to determine how long a statement should be, then that symbol must be defined prior to its use. An example of this is the RMB statement; one must have already defined "XXX" in order to assemble "RMB XXX". Other statements which require their operands to be previously defined include BZS, EQU, SETDP, etc. If a symbol used in one of these statements is not defined, the error "FOR" is generated (meaning "forward referenced symbol"). This error is generated whether or not the symbol is eventually defined.

A statement such as "LDA XYZ,X" could be assembled with a length of two, three, or four bytes. If XYZ has been previously defined, then the shortest possible length will be used. If XYZ is not defined until later in the program, however, the longest form of the instruction will be used. This is because, as with the RMB example above, it is necessary to know the length of each instruction during the first pass.

***** REGISTER LISTS

The instructions EXG, TFR, PSHS, PSHU, PULS, and PULU require register lists as operands. The register names must be from the list D, X, Y, U, S, A, B, CC, DP, and PC, and must be separated by commas. EXG and TFR require two registers, and the others may have any number. The assembler allows statements such as TFR A,X whose results are undefined in the 6809 hardware. PSHS S is meaningless; it is assembled as PSHS U since S and U occupy the same bit in the object code operand.

***** SHORT VERSUS LONG ADDRESSING

There are many cases where an instruction can be assembled in either a long or a short form. Examples:

```
LDA XYZ        if XYZ < $100, then direct addressing may be used
STA PDQ,X      this may take 2, 3, or 4 bytes, depending on PDQ
LDX [ABC,Y]    this may take 3 or 4 bytes
```

It is necessary that the assembler decide on the length of the instruction during the first pass, so that all subsequent symbols may be defined correctly. If all symbols used in such an instruction have already been defined, then the shortest addressing mode may be used. It is therefore a good idea to define variables at the top of the program instead of at the bottom. Where there is an undefined symbol in an expression, the longest form of the instruction will be used by default.

If the programmer knows which mode should be used, the symbols "<" and ">" may be used to specify it. Examples:

```
LDA <XYZ       use direct addressing
STA <PDQ,X     use 8 bit offset (not 5 bit unless PDQ defined)
LDX >[ABC,Y]   use 16 bit offset
```

The use of direct addressing is dependent not on the expression being less than $100 (as we sometimes imply) but in the high byte being equal to the assumed value of the DP register. It is the programmer's responsibility that the value in the actual DP register at run time equals the high byte of the address. The value assumed by the assembler for use in deciding if direct addressing is appropriate is set by the SETDP pseudo op; SETDP is followed by an expression which should evaluate to an 8-bit value which will be assumed to have been loaded into the DP at run time.

EXAMPLE:

```
LDA NERF              This uses extended addressing
LDB #NERF/$100        Get the high byte of address of NERF
TFR B,DP              Set the hardware DP register
SETDP NERF/$100       Let the assembler know about it
LDA NERF              This then uses direct addressing
```

***** MNEMONICS

All of the standard 6809 mnemonics are accepted. 6800 mnemonics are accepted also, and cross assembled to 6809 code. (The art of cross assembling 6800 programs is described in a section of the chapter on assembly language.) There are some new pseudo-ops which should be noted, including conditional assembly, local labels, and listing suppression.

Instructions which require the register A or B as operand may be written in either of two ways: CLR A or CLRA; ADD A XYZ or ADDA XYZ.

Although the editor tabs the mnemonic field to column eight, the assembler does not require this. The mnemonic may start anywhere after column one as long as it is preceded by a space. This gives another six columns to the operand field where it is necessary.

***** DATA GENERATION

The FCB pseudo-op is used to place data bytes into the object code. It is followed by one or more expressions, separated by commas. Each expression which does not evaluate in the range -128 through 255 generates the ERROR - BYT message and is truncated.

The FDB pseudo-op places two-byte values into the object code. Any expression is valid, since all arithmetic is done as two-byte values in the expression processor.

The FCC (Form constant character) statement has two forms. The first has for an operand a single string which is delimited by some non-numeric character. The second form has an expression which determines the length of the string. In no case will the string be allowed to extend beyond the line boundary; see examples below.

```
FCC  "THIS IS THE STRING"
FCC  /NO END SLASH NEEDED
FCC  4,NERF
FCC  100,
```

The expression in the second form of the FCC statement must start with a digit (1-9) to distinguish it from the first form. Note that zero is not allowed, as even Motorola has produced programs containing such statements as:

```
FCC  0...............0
```

If an expression more general than a single number is required, it must start with a number anyway. To use the symbol "LENGTH" as the length, for example, you could do it like this:

```
FCC  1*LENGTH,THIS IS THE STRING
```

The RMB statement is used to reserve bytes for variables. It does not necessarily generate any object output; its function is to update the assembler's PC. If it appears between sections of generated object, however, and the output is to cassette tape, output must be generated and so the area is set to $3F (SWI).

The BZS statement is like RMB in syntax but always sets the area to zeroes. Its use is not recommended as a way of presetting variables, as such a program must be reloaded whenever it is to be rerun. It is included for compatibility with other assemblers.

***** CONDITIONAL ASSEMBLY

Programmers are commonly faced with the problem of having to maintain two or more versions of a program. Common examples include: (1) A stripped-down version versus a full-blown version; (2) Versions which run on two versions of a computer; (3) Versions supporting different kinds of I/O. If different sources are kept, it becomes difficult to be sure that changes made to one version are made identically to all versions. The best solution to this problem is conditional assembly.

EXAMPLE:

```
TRS80C EQU -1          set not true for this assembly
       ...
       IFEQ TRS80C
OUTPUT JMP $A30A       output routine for 80C
       ELSE
OUTPUT JMP $D286       output routine for other system
       ENDC
```

Thus lots of small changes may be made just by changing one line at the beginning of the program; and all differences are thoroughly documented.

The above discussion is not intended to imply that this is the only use for conditional assembly; it is a powerful tool which a programmer will find helpful in many situations.

The statements IFEQ, IFGE, IFGT, IFLE, IFLT, and IFNE are used to define the start of a conditional segment. The mnemonic is followed by a single expression which is evaluated in 16-bit arithmetic. If the condition specified in the mnemonic (=0, >=0, >0, <=0, <0, or <>0 respectively) is true, then assembly proceeds; otherwise no object code is generated (and no symbols defined) until the end of the conditional block. The condition may be reversed with the ELSE pseudo-op, which causes assembly either to restart or to be suspended. The conditional block is terminated with a ENDC statement.

IFs may be nested to a virtually unlimited depth. A sequence of IFxx-ENDC or IFxx-ELSE-ENDC will be ignored if it is within a block that is not being assembled; it will be processed normally if it is within a block that is being assembled.

Something to think about:

```
       1FNE *!.$FF
       RMB 256-*!.$FF
       ENDC
```

***** LOCAL LABELS

Programs commonly get filled with many symbol names which have meaning only in the immediate vicinity of their definition. Examples include

the labels used to branch over one or two statements or to form a small loop for setting or moving data. Over half of the symbols in a program may fall into this category.

There are four possible solutions:

(1) Keep thinking of creative, descriptive symbol names. This will strain the creativity and memory of any programmer.
(2) Use some major label nearby, and tack on a number. (EG, PUTIT, PUTIT2, PUTIT3, ...) This works fairly well but fills up a symbol table listing with so much meaningless repetition that it is hard to read.
(3) Use statements like BNE *-9. People who do this should be tarred and feathered.
(4) Use local labels.

In this assembler, a local label is any single letter followed by an at-sign ("@"). It is defined only within a block bounded by blank lines. It is not listed in the symbol table. Each of the 26 local labels may be reused any number of times.

Example:

```
MOVE    LDX #START
        LDY #WHERE
A@      LDA ,X+
        STA ,Y+
        CMPX #END
        BLO A@

        TST NULLIT
        BEQ A@
        CLR ,Y+
A@      RTS
```

The use of blank lines to delimit the scope of a symbol is not intended to cause the programmer to think of blank lines as a form of pseudo-op. Blank lines should be sprinked around a program in any case; a readable program will have a blank line about every dozen lines. The logical division of a program into sections (such as the two sections in the subroutine above) which is defined by the blank lines corresponds exactly with the intended scope of the local variables. It is NOT intended (in spite of the example above) that local variables be reused in close proximity; if enough space is left between incarnations of local labels, the effect of blank lines on the labels should not need to concern the programmer.

***** LISTING SUPPRESSION

The pseudo-ops NLST and LIST are used to suppress assembly Generally a NLST is put before a table or other dull area, and LIST after it, so that these areas need not be listed every time. NLSTs may be nested; if a subroutine is bracketed by NLST then listing will not resume until after its last LIST even if it contains

NLST/LIST pairs. A LIST then NLST will cause listing in an otherwise unlisted area. A LIST at the beginning of the entire program will cause listing of all NLST areas except those which are nested.

The mechanism behind NLST/LIST is simple: NLST decrements a counter which LIST increments. Listing only occurs when the count is positive.

Lines which generate error messages are always listed.

SECTION 4 - THE ABUG MONITOR

The ABUG monitor is a program which allows direct access to memory and registers of the color computer, and is used to oversee the execution of programs during debugging. It is similar to the CBUG monitor, but has been tailored for use with programs produced by SDS80C; in particular it allows access to the assembler's expression processor and through that to the symbol table.

ABUG is called from the assembler when an object program has been written to memory, and may also be called from the editor with the command "@=" (return). When called, it displays its prompt "ABUG: " to the screen.

When the monitor issues its prompt, a single-letter command may be typed, along with any parameters needed. With the exceptions listed below, parameters are in hex and any non-hex character (including backspace) will cause the command to abort and return to the monitor prompt level. Commands are shown as they appear on the screen, with spaces between the parameters, but these spaces are added by the monitor and should not be typed in by the user.

The following are the commands:

G Go. This will execute the object program. The registers, including the PC, are loaded from the register list as displayed by the R command. This will start the object program at the beginning, or, if it was suspended by an SWI instruction, restart it at that point. (See section on the stack frame for more details.)

M Memory examine & change. "M 1234" will display a line of eight bytes, with the cursor by address 1234. The cursor may be moved up, down, left, or right with the arrow keys (or the space bar) to display more memory. Typing hex numbers will enter data into memory. A carriage return will exit the command. Another "M" has the same effect as a carriage return followed by an "M". Inverted numbers indicate a write to an area where there is no RAM.

? Evaluate expression. This calls the assembler's expression evaluator. Various examples of its use:

```
? NERF              prints the value of symbol NERF
? 1234              converts 1234 to hex
? $12*($317+$3F)    may be used as a hex calculator
? *+7               this adds 7 to the previous result
? NERF-(*+3)        or any combination of the above
```

If no answer is given, a syntax error has been found. If a question mark is given as the answer, there was an undefined symbol. Use of the "U" command invalidates the symbol table.

R Display register list. This shows what is in the stack frame, which is what will be loaded into the registers by the G or J commands. (See section on stack frame for more details.)

T Transfer block of memory. "T 0123 1234 2345" will copy the contents of locations 0123 through 1234 (inclusive) to the area of memory starting at 2345. Also note that the command "T 0123 1234 0124" will copy the contents of 0123 throughout the block of memory 0124 through 1235.

J Jump to machine-language subroutine. The registers A, B, X, Y, and U are loaded out of the register list shown by the R command, and a JSR instruction is done to the location specified by the parameter. "J 1234" will jump to location 1234, and when an RTS is encountered control will return to the monitor.

C Change register contents on stack. This is followed by a single-letter parameter which designates which register to change. Admissible register names are A, B, X, Y, U, D (direct page), C (condition codes), and P (program counter). This transfers control to the memory examine / change function ("M") at the address on the stack where the specified register is stored.

S Save to cassette. "S 1234 2345 1357 MYFILE" (followed by a carriage return) will write a file on cassette with the name MYFILE and containing the data from 1234 through 2345 inclusive. 1357 is the transfer address which is loaded into the EXEC pointer. The tape can be loaded with BASIC's CLOADM command and if an EXEC is then done control will transfer (in this example) to location 1357.

L Load from cassette. "L 1234 MYFILE" (return) is the same as the command in BASIC: CLOADM"MYFILE",1234. The 1234 is the offset; normally this parameter is zero. This command may be used to load tapes produced by the "S" command or by the assembler. In both the L and S commands, backspace is allowed while entering the filename.

U Unstack. This resets the stack pointer to the point where an RTS in the user program will return directly to the editor. This command may be used when in danger of a stack overflow. It will remove the symbol table from the stack, freeing that memory but making the symbols unavailable to the "?" command.

* Return to the editor. It is assumed that locations $0000 through $000F have not been changed. Also, if the loading or execution of object programs has overwritten the text buffer, Read or Delete should be used to clear it out.

The stack frame

The R command displays the last 12 bytes on the stack, which is called the "stack frame". The "G" command is really an RTI instruction which loads those 12 bytes into the registers of the CPU so that execution resumes at the location in the bytes labelled PC. If an SWI instruction is encountered, this stacks the registers again

and returns control to the monitor.

When ABUG is entered, the registers are zeroed except for the PC, which contains the address of the start of the program in memory. If it is desired to start elsewhere, use the "CP" command. (Note: all other registers are zero except for CC, which contains necessary flags.)

In the process of debugging a program, the programmer adds SWI instructions (called "breakpoints") into various critical parts of the program. As these executed, the contents of the registers at that point are printed out and the program may be restarted with a "G". The operator then has the option to examine or change memory or registers at those points. Other SWIs may be added using ABUG; these are put in place of other opcodes and must be removed (and the PC decremented) before hitting "G".

If the "J" command is used, this has the effect of making a subroutine call from ABUG; the RTS at the end of the subroutine will return control to ABUG. If an SWI is encountered in the subroutine, then ABUG is entered AT ANOTHER LEVEL. This is worth understanding. A "G" at this point will return control to the subroutine, and only when that subroutine executes an RTS will ABUG be reentered at its main level. That RTS will restore the stack frame to the set of values which were displayed before the "J" command was given.

It is not intended that SWI be used as a halt instruction. A program being debugged should end in either an RTS (which will return to the editor) or a SWI followed by LBRA START. In the latter case the program may be restarted any number of times with "G".

The number of levels on which ABUG may be entered is limited only by the size of the stack. However, if the "J" command is repeatedly used to execute a routine which ends in SWI instead of RTS a stack overflow will eventually occur. The "U" command may be used to clean unwanted levels off the stack; this will also delete the symbol table.

The "*" command may be used to return to the editor from any level of ABUG.

An RTS in the target program will return to the editor, unless it was called from the "J" command in which case it returns to ABUG.

The idea of levels of ABUG is best understood by remembering that ABUG is called BY THE USER PROGRAM; it may also CALL a part of the user program but this is not the normal method of execution. When ABUG is first entered from the assembler, it is set up as if a SWI instruction had been assembled as the first instruction. If ABUG calls a user program, and then that program calls ABUG (via SWI), then ABUG has been entered recursively.

MAP OF LOW MEMORY WHILE EXECUTING A USER PROGRAM:

```
0000  Start of text buffer
0002  End of text
0004  End of text buffer
0006  Number of spaces for editor's tab
0007  =$60 for tabs; =$FF to disable
0008  constant =$0420
000A  constant =$05E0
000C  start of symbol table
000E  end symbol table, save stack
0010  last value from "?" command
```

During execution of the editor and the assembler, other locations throughout page zero are used, but these need not be saved by user programs.

SECTION 5 - THE 6809 ASSEMBLY LANGUAGE

This section gives a brief description of the language used for programming the 6809 microprocessor which runs the Color Computer. A complete description of the language is the subject of several books; only a brief summary will be given here.

***** ARCHITECTURE

The 6809 has a 16-bit (two byte) accumulator register called "D". The upper (most significant) byte of "D" is called the "A" accumulator and the lower (least significant) byte is called the "B" accumulator.

There are four 16-bit index registers, one of which is the system stack pointer. "X", "Y", and "U" may be used for any purpose, while "S" should remain pointing to a stack area for use with interrupts and subroutine calls.

There is an eight-bit condition code register ("CC") in which each bit has meaning. From the left (MSB), these bits are:

- E - "entire" - all registers on stack
- F - "fast IRQ mask" - this bit inhibits fast interrupts
- H - "half carry" - set by ADD and used by DAA
- I - "IRQ mask" - this bit inhibits normal interrupts
- N - "negative" - result of load, store, or operation was negative
- Z - "zero" - result of load, store, or operation was zero
- V - "overflow" - 2's complement signed overflow on ADD or SUB
- C - "carry" - unsigned carry or borrow, or result of shift

There is an eight-bit direct page register ("DP") which contains the upper byte of the address to use with direct mode addressing. This register ordinarily contains zero and does not need to be used.

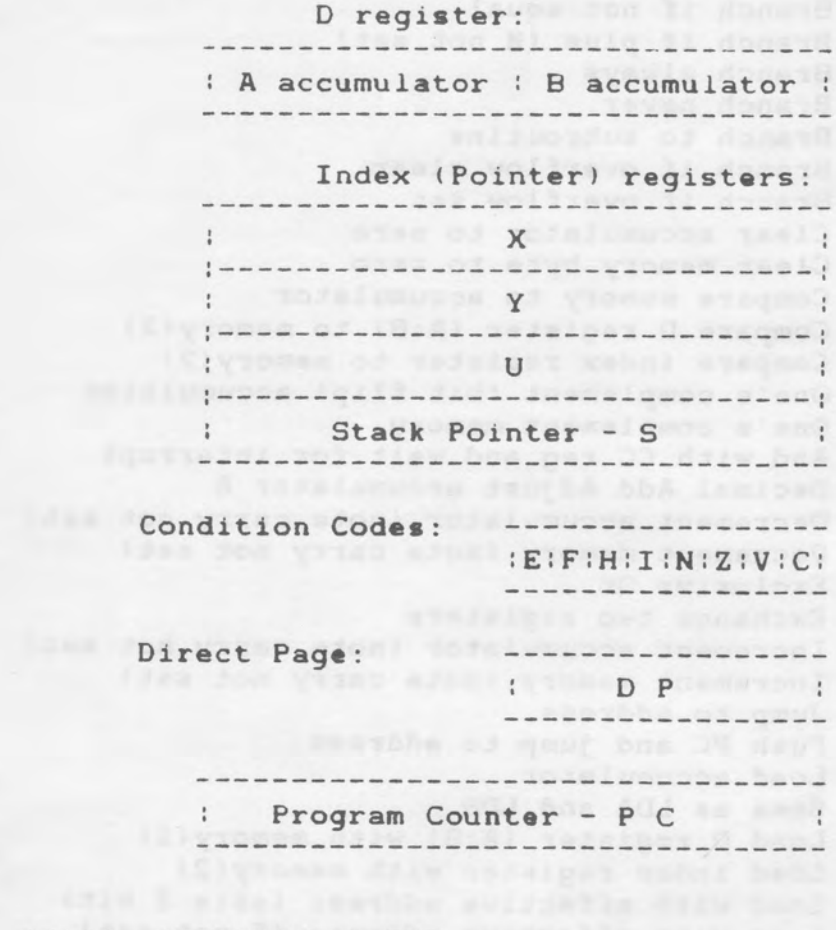
Finally, there is, of course, a 16-bit Program Counter ("PC").

Each 16-bit address corresponds to an eight-bit location in the address space. This location could be in RAM (read/write memory), ROM (read only memory), or I/O areas, and "memory reference" instructions may have different effects accordingly. The instruction "LDA $FF02" might be used not to load "A" but to acknowledge an interrupt. There are no distinct I/O instructions in the 6809.

When 16-bit (two byte) values are stored in memory, the most significant byte is stored in the lower address. This is the opposite of how it is done on inferior processors such as the 6502 and the Z80. In the description of the instruction set below, the notation "memory(2)" refers to two bytes of memory used to store such a value.

The bit numbering convention for the 6809 is that the least significant bit (the rightmost bit) is bit zero; the most significant bit is bit 7 (for one-byte values and registers) or bit 15 (for two-byte values and registers).

***** 6809 PROGRAMMING MODEL



***** INSTRUCTION SET

ABX	Add B to X (Unsigned)
ADCA or ADCB mem	Add memory and carry to accumulator
ADDA or ADDB mem	Add memory to accumulator
ADDD mem	Add memory(2) to D register (A:B)
ANDA or ANDB mem	Logical And memory to accumulator
ANDC #value	Same as ANDCC
ANDCC #value	Logical And Immediate with condition codes
ASLA or ASLB	Arithmetic shift left accumulator
ASL mem	Arithmetic shift left memory
ASRA or ASRB	Arithmetic shift right accumulator
ASR mem	Arithmetic shift right memory
BCC label	Branch if carry clear
BCS label	Branch if carry set
BEQ label	Branch if equal set
BGE label	Branch if greater or equal (signed)
BGT label	Branch if greater than (signed)
BHI label	Branch if higher (unsigned)
BHS label	Branch if higher or same (unsigned)
BITA or BITB mem	Bit test with memory (And but save reg.)
BLE label	Branch if less than or equal (signed)
BLO label	Branch if lower (unsigned)
BLS label	Branch if lower or same (unsigned)

```
BLT  label                      Branch if less than (signed)
BMI  label                      Branch if minus (N set)
BNE  label                      Branch if not equal
BPL  label                      Branch if plus (N not set)
BRA  label                      Branch always
BRN  label                      Branch never
BSR  label                      Branch to subroutine
BVC  label                      Branch if overflow clear
BVS  label                      Branch if overflow set
CLRA or CLRB                    Clear accumulator to zero
CLR  mem                        Clear memory byte to zero
CMPA or CMPB  mem               Compare memory to accumulator
CMPD  mem                       Compare D register (A:B) to memory(2)
CMPS, CMPU, CMPX, CMPY  mem     Compare index register to memory(2)
COMA or COMB                    One's complement (bit flip) accumulator
COM  mem                        One's complement memory
CWAI  #value                    And with CC reg and wait for interrupt
DAA                             Decimal Add Adjust accumulator A
DECA or DECB                    Decrement accumulator (note carry not set)
DEC  mem                        Decrement memory (note carry not set)
EORA or EORB  mem               Exclusive Or
EXG  reg, reg                   Exchange two registers
INCA or INCB                    Increment accumulator (note carry not set)
INC  mem                        Increment memory (note carry not set)
JMP  mem                        Jump to address
JSR  mem                        Push PC and jump to address
LDA or LDB  mem                 Load accumulator
LDAA or LDAB  mem               Same as LDA and LDB
LDD  mem                        Load D register (A:B) with memory(2)
LDX, LDY, LDU, LDS  mem         Load index register with memory(2)
LEAX, LEAY  mem                 Load with effective address (sets Z bit)
LEAU, LEAS  mem                 Load with effective address (Z not set)
LSLA or LSLB                    Logical shift left (same as ASL)
LSL  mem                        Logical shift left (same as ASL)
LSRA or LSRB                    Logical shift right (zero fill)
LSR  mem                        Logical shift right (zero fill)
MUL                             Multiply A x B, result in D
NEGA or NEGB                    Negate (2's complement)
NEG  mem                        Negate memory byte
NOP                             No operation
ORA or ORB  mem                 Inclusive Or
ORAA or ORBB  mem               Same as ORA and ORB
ORCC  #value                    Or to condition codes register
PSH  reg,reg,...                Same as PSHS
PSHS  reg,reg,...               Push registers to system stack
PSHU  reg,reg,...               Push registers, using U as stack pointer
PUL  reg,reg,...                Same as PULS
PULS  reg,reg,...               Pull registers from system stack
PULU  reg,reg,...               Pull registers, using U as stack pointer
ROLA or ROLB                    Rotate left through carry
ROL  mem                        Rotate left through carry
RORA or RORB                    Rotate right through carry
ROR  mem                        Rotate right through carry
RTI                             Return from interrupt
RTS                             Return from subroutine (pull PC)
```

SBCA or SBCB mem	Subtract, and subtract carry
STA or STB mem	Store accumulator
STAA or STAB mem	Same as STA and STB
STD mem	Store D register (A:B) to memory(2)
STX, STY, STU, STS mem	Store index register to memory(2)
SUBA or SUBB mem	Subtract memory from accumulator
SUBD mem	Subtract memory(2) from D register (A:B)
SWI, SWI2, SWI3	Software interrupt
SYNC	Halt until interrupt (masked or not)
TFR reg,reg	Transfer reg to reg
TSTA or TSTB	Test for zero and negative
TST mem	Test for zero and negative

***** ADDRESSING MODES

In the list of instructions above, the word "mem" in the left column indicates that a memory address should be specified with the instruction. These memory addresses should be in the one of the following forms:

#expression	immediate addressing mode - the value given is used as an operand. This mode is required for instructions such as ANDCC, ORCC, and CWAI. It is not allowed for memory-modifying instructions such as stores, shifts, etc.
expression	extended or direct mode is used by the assembler - the value given is used as an absolute address.
[expression]	indirect - the value given is used as the address of the address of the operand.
expression,index	indexed - where "index" is one of the registers X, Y, U, S, or PC.
,index	indexed - same as 0,index.
,index+	auto increment - the index register X, Y, U, or S is incremented after use.
,index++	auto increment - the index register is incremented twice after use.
,-index	auto decrement - the index register X, Y, U, or S is decremented before use.
,--index	auto decrement - the index register is decremented twice before use.
acc,index	accumulator offset - "acc" is A, B, or D and is used as a signed offset to X, Y, U, or S.
expression,PCR	PC relative - the address specified by the value of the expression is referenced by means of the PC indexed mode.
[expression,index]	
[,index]	
[,index++]	
[,--index]	
[acc,index]	
[expression,PCR]	indirect indexed modes

***** 6800 CROSS MNEMONICS

The following mnemonics are supported by the assembler so that code written for the 6800 processor can be assembled. See the paragraph elsewhere in this section for details of 6800 cross assembling. Some of these instructions (such as INX or CLI) are clearer and easier to use than their 6809 equivalents and may be used instead.

6800 inst	Generated code equivalent
ABA	PSHS B; ADDA ,S+
CBA	PSHS B; CMPA ,S+
CLC	ANDCC #$FE
CLI	ANDCC #$EF
CLV	ANDCC #$FD
DES	LEAS -1,S
DEX	LEAX -1,X
INS	LEAS 1,S
INX	LEAX 1,X
SBA	PSHS B; SUBA ,S+
SEC	ORCC #$01
SEI	ORCC #$10
SEV	ORCC #$02
TAB	TFR A,B; TST A
TAP	TFR A,CC
TBA	TFR B,A; TST B
TPA	TFR CC,A
TSX	TFR S,X
TXS	TFR X,S
WAI	CWAI #$FF

Also included are some 6800-like mnemonics which relate to Fast Interrupts, which the 6800 does not have. They may be clearer to use than their 6809 equivalents and are included for this reason.

CLF	ANDCC #$DF	Clear Fast Interrupt inhibit bit
CLIF	ANDCC #$AF	Clear both interrupt inhibit bits
SEF	ORCC #$40	Set Fast Interrupt inhibit bit
SEIF	ORCC #$50	Set both interrupt inhibit bits

***** PSEUDO OPS

The following are the pseudo op mnemonics which are accepted by this assembler. The data generation pseudo ops are standard for the 6809; the assembler control pseudo ops are given specifically for this assembler.

BSZ - BLOCK ZERO STORE (Like RMB only sets the area to zero)
ELSE - (See section on conditional assembly)
END - END OF PROGRAM (May be followed by transfer address)
ENDC - END CONDITION (See section on conditional assembly)
EQU - EQUATE SYMBOL
FCB - FORM CONSTANT BYTES
FCC - FORM CONSTANT CHARACTERS

```
FDB  - FORM DOUBLE BYTES
IFxx - CONDITIONAL ASSEMBLY (xx = EQ, NE, GT, GE, LT, LE)
LIST - RESTART LISTING
NAM  - NAME OF PROGRAM (Sets object tape name)
NLST - SUSPEND LISTING
OPT  - OPTIONS (No effect; for compatibility only)
ORG  - SET ORIGIN
PAGE - (No effect; for compatibility only)
RMB  - RESERVE MEMORY BYTES
SETDP- SET ASSUMED DIRECT PAGE REGISTER
SPC  - SPACE LISTING (Leave blank lines)
TTL  - TITLE (Same as NAM)
```

***** POSITION INDEPENDENT CODE

The 6809 programmer should understand position independent code, as it is a powerful tool which is readily available on this processor. Although it is possible to write P.I.C. on other CPU's such as the 6800, it is seldom easy. On the 6809, however, there is almost no excuse not to.

Position independent code (or P.I.C.; relocatable code; or run-anywhere code) is code which can be moved to anywhere in the memory space AFTER ASSEMBLY and correctly run. For example, a P.I.C program which runs at $C000 could be block moved up to $D000 and it would run just as well. If it contained any statement such as JMP LABEL (where LABEL is within the program) then it would not be P.I.C., since it would jump to the wrong address if the code were moved. Writing P.I.C., then, is basically avoiding certain statements and addressing modes which do not work when the object code is moved.

The most obvious rule is this: Replace JMP with LBRA (long branch), and replace JSR with LBSR (long branch to subroutine). In itself, however, this is not enough; a program which is only partially P.I.C. is not P.I.C. at all and is merely slower than the version which contains JMPs. Therefore, the discussion below is included in order to allow you to write code which is entirely P.I.C.

All symbols should be classified as to whether they are in the program or absolute; that is, whether or not they will move when the program does. A label is in the program; so is a variable which is declared as part of the program (which makes it non-ROMable, but that's another story). A variable in page zero is absolute. A routine in the BASIC ROM is absolute. A symbol defined by an equate statement is absolute, unless there is a program label on the right side of the equate. A variable in a stack frame is absolute (which is yet another story).

Any reference to a symbol in the program must be relative; any reference to an absolute symbol must be absolute. A jump to a label in the program must use BRA or LBRA, while a jump to a routine in ROM cannot use LBRA and must use JMP.

A reference to a variable or constant which is part of the program

must use the PCR addressing mode. LDA XYZ,PCR has the same effect as LDA XYZ except that relative addressing is used. On the other hand, the PCR mode may not be used when referencing variables which are defined by absolute symbols. Remember the adage, "Everything not mandatory is prohibited."

The statement "LDX #TABLE" assembles to an absolute address which must not be used if TABLE is within the program. The statement to be used instead is "LEAX TABLE,PCR". A thorough understanding of the LEA statement in its many forms is necessary for a good 6809 programmer.

The statement "CMPX #TABLE" is harder to replace. It is often used to check to see if we have finished stepping through a table. It is best avoided by various means which depend on the program in question, but if a quick fix is needed, this sort of thing will work:

```
        LEAX ENDTAB,PCR
        PSHS X
        LEAX TABLE,PCR
LOOP    ADDA ,X+    (or whatever)
        CMPX 0,S
        BLO LOOP
        PULS X      (clean up stack)
        ...
TABLE   FCB 1,2,3,4,5
ENDTAB  EQU *
```

In the example above, by the way, the PULS X could have been replaced by a LEAS 2,S. However, it is clearer and less conducive to errors to match a pull with a push.

One source of problems is the FDB statement. Any time a label in the program appears in an FDB statement, an absolute constant may be generated. A typical example is the jump table, wherein this sort of thing goes on:

```
        LDX #TABLE
        ASL B
        LDX B,X
        JMP 0,X
TABLE   FDB PLACE1,PLACE2,PLACE3,...
```

The LDX becomes an LEAX, of course, but what about the FDB? One solution is to add onto X (just before the jump) some relocation constant which can be obtained by the strange-looking statement LEAX 0,PCR. Another solution is this:

```
        LEAX TABLE,PCR
        ASL B
        ABX
        LDD 0,X
        JMP D,X
TABLE   FDB PLACE1-*,PLACE2-*,PLACE3-*,...
```

Who says you can't have a "JMP" in P.I.C.?

***** 6800 CROSS ASSEMBLY

The 6809 processor in the Color Computer is, according to Motorola who designed it, upward compatible on a source code level with its predecessor, the 6800. This means that any 6800 program may be typed into the SDS80C and assembled and run.

In actuality, the converting of 6800 programs to run on a 6809 is an art. This appendix will give you enough information to make most programs run, but cannot make the process foolproof or elegant.

Many 6800 statements do not need to be converted. The statement CBA (Compare B to A), for example, which generates a single instruction on a 6800, will generate a "PSHS B; CMPA ,S+" in this assembler, which accomplishes the same thing. The only thing to watch out for is a 6800 program which does not keep a stack; if the S register is used as an index register then the converted CBA will alter a byte of data. Such a program, however, is not worth converting.

Some of the 6800 statements are worth using in new 6809 programs. These are CLC, CLI, CLV, DEX, INX, SEC, SEI, and SEV. These are much clearer than their 6809 counterparts. They are accepted by any 6809 assembler which conforms to Motorola's standards and there is no reason not to use them.

The following problems will creep in when cross-assembling:

The condition codes are not set identically. The CPX instruction on the 6800 (which assembles as a CMPX) does not affect the carry bit. If carry is set as a flag before a CPX the 6809 version will not work.

The TST (test) on the 6809 does not clear carry the way it did on the 6800. This is generally not a problem, but the sequence TST VAR1 / BHI LABEL will not work.

Right shifts on the 6809 do not affect the overflow bit, so the sequence LSR A / BVS LABEL will not work. Such a test would, however, be rare.

Use of the H-flag for other than its intended purpose ("DAA") may not yield identical results. And, of course, any assumption by a 6800 program that E and F bits are always one will not be true.

The stack is the other big problem. Any program which deals with an interrupt stack frame (such as a monitor like ABUG) will not cross assemble since the order and size of the frame is different. Any program that contains an LDS or STS is suspect.

The 6800 instruction TXS was actually a "transfer X to S and decrement"; TSX likewise incremented. Although this could have easily been incorporated in the cross-assembled code, it is more accurate to leave out the decrement/increment. The reason is this: The 6800 stack pointer pointed at the next free byte; the 6809 stack pointer points at the first used byte. The instruction TSX will point the X register at the last byte pushed if it increments on the 6800

but does not increment when cross-assembled to the 6809.

The consequence of the considerations in the paragraph above is simply this: A 6800 program which contains only TSX and TXS instructions is safe to cross assemble. Any LDS or STS instruction may need work. An LDS #$xxxx (such as is commonly found) may have its value increased by one, such as from $3FFF to $4000. An STS SAVSTK followed later by LDS SAVSTK is all right, but if LDX SAVSTK appears it is guaranteed not to work.

The last problem is in software timing loops. These have to be looked at anyway since they depend on processor speed (0.895 MHz on the Color Computer). Many 6809 instructions take more cycles than the 6800 equivalents. Load A extended takes five instead of four cycles. Compare B to A (which crosses to the PSH/CMP shown above) will take twelve instead of two cycles. Conditional branches, however, only take three cycles, so the loop

```
        LDX #COUNT
LOOP    DEX
        BNE LOOP
```

will still take eight cycles as it did on the 6800, even though both of its instuctions take a different number of cycles each. Most other timing loops will require careful counting.

In summary, to cross assemble a 6800 program, look for these items:

1. CPX - is carry tested afterwards?
2. TST - is carry tested afterwards?
3. right shifts - is overflow tested?
4. any unusual operations on the condition codes
5. all LDS and STS instructions
6. any software timing loops

If all of these points are checked, most 6800 programs should give little trouble in cross assembly.

APPENDIX 1 - MEMORY FULL CONDITIONS

Since there is a finite amount of RAM in the computer, any programmer with more than a finite amount of creativity will sooner or later write a program which overflows memory. This overflow condition can occur while editing, loading, building a symbol table, emitting object code to memory, or executing the program. The way in which the SDS80C handles these various conditions is given below.

It is best not to try to work with a text file which continually pushes the limit of available memory. Although every effort is made by the SDS80C to accurately recover from memory overflows, it is always possible to get unexpected results. At the first sign of full memory, delete some comments or shorten some variable names to make more room. If you have to, split off some subroutines to be assembled separately and loaded as object tapes.

The top of memory used by SDS80C is determined by the stack pointer upon entry. The sequence "LDS #$xxxx / JMP $C000" will restart the editor with all RAM above $xxxx reserved. Since there is ordinarily about 200 bytes reserved above the stack, the sequence "LDS #$4000 / JMP $C000" will make another 200 bytes available to SDS80C. (For 32 K systems, the $4000 above would be $8000 instead.)

Overflows while reading: If memory is full during a cassette read, reading stops and the last portion read is displayed on the screen. Whenever reading a cassette, make sure that when the tape stops the end of the program is visible; if it is not, a read error or a memory full condition has occurred.

Overflows while editing: The number in the upper right corner of the screen indicates remaining memory space. If it goes to zero, memory is full or near full. In this situation, an attempt to enter Line Insert, eXchange, Change, etc., will cause the display "*** FULL ***" and require the Break key to return to command mode.

Overflows in Assembly Pass 1: If there is not enough room for the symbol table, the message "ERROR - OVF" will be displayed. Press ENTER to return to the editor.

Overflows in code generation: If there is not enough room for object code between the text file and the symbol table (and option M is in effect) the message "ERROR - OBJ" will be displayed. Assembly may continue, but no more code will be generated. It is possible in some cases to overwrite the last few bytes of code without generating the message; this would occur only with unusual stack usage.

Overflows in program execution: If there is any question about having sufficient stack room left for the target program, the command "U" may be used in ABUG to remove the symbol table from the stack. If a stack overflow does occur during execution, the last bytes of the object program get overwritten. Obviously, no error message is generated since the user program is in control at this point.

APPENDIX 2 - BASIC ROM ENTRY POINTS

Entry points for I/O in the Color Computer's ROM are given below. This is intended only to provide information necessary for standard I/O; a complete description of the routines available is beyond the scope of this manual. For further information about the ROM and a description of the hardware-level I/O of the Color Computer, the Micro Works has available a source generator program and an information package which enables the user to produce a complete source listing of the ROMs as well as providing hardware information.

RAM extends from 0 through 0FFF (4K), 3FFF (16K), or 7FFF (32K). The screen is normally at 0400 through 05FF, and output may be done simply by writing to this area. The Basic ROM is at A000 through BFFF. The Extended Basic ROM is at 8000 through 9FFF. The area C000 through FEFF is available to the cartridge slot, and the SDS80C lives at C000 through DFFF.

There is a PIA (Peripheral Interface Adapter) at FF00 which is used to scan the keyboard, and another at FF20 which controls all other I/O. Above this address are various registers which control the functions of the SAM chip (which sends RAM data to the video display, refreshes the dynamic RAM, etc.)

Useful calls to the ROM:

```
JSR $A30A  write character in A reg to screen
JSR $A2BF  write character in A to printer
JSR $A1B1  wait for key press and read keyboard
JSR $A1C1  read keyboard; return zero if no new key
JSR $A393  read line from keyboard; return X = one less than the
           address of the buffer; zero byte at end of buffer
JSR $A9DE  read joysticks.  Leave the four resulting values in
           $015A through $015D.
```

There is no good call to print an ASCII string. It is best to include the following subroutine in your own code:

```
PSTRNG LDA ,X+
       BEQ A@
       JSR $A30A
       BRA PSTRNG
A@     RTS
```

To return to the editor after running an object program, the first $10 bytes of memory should be preserved. The ABUG monitor may access the next 8 bytes, so they should be avoided. The lowest address references by the I/O routines above is at $006C, so the area from $0018 through $006B may be used by user programs. Because of the Direct Page register in the 6809, however, zero page need not be used for direct page addressing at all; it should be noted, however, that the direct page register should contain zero whenever calling the ROM routines above.

APPENDIX 3 - TIMING LOOPS

It is often necessary to write programs in which a certain routine must execute at a certain time or after a certain delay. Any program dealing with serial I/O, sound output, interactive graphics, or mechanical delays must involve timing or synchronizing loops.

The processor speed on the Color Computer is 0.895 MHz. A one millisecond timing loop is:

```
        LDX #111
        NOP
        NOP
LOOP    LEAX -1,X    (Same as DEX)
        BNE LOOP
```

where the NOPs are there simply to bring the total up to exactly 1000 uSec.

It is possible to synchronize to the 60 Hz vertical refresh rate. The following program allows a section of code to be done sixty times each second:

```
        LDA #$34     Interrupts Off
        STA $FF03    To PIA control register
        LDA $FF02    Clear Flag
LOOP    LDA $FF03    Check control register
        BPL LOOP     Wait til refresh has occurred
        LDA $FF02    Clear Flag
* Do whatever you want in here
        BRA LOOP     Go sync again
```

The following routine syncronizes to the 63 uSec horizontal interrupt. Due to the high speed of this signal, the SYNC instruction is used.

```
        ORCC #$50    Inhibit IRQ and FIRQ
        LDA #$35     Interrupts enabled to processor
        STA $FF01    To PIA control register
        LDA $FF00    Clear Flag
LOOP    SYNC         Wait for inhibited IRQ
        LDA $FF00    Clear Flag
* Do whatever you want in here
        BRA LOOP     Go sync again
```

For a very short delay, try the MUL instruction. It takes 11 cycles (about 10 uSec) in only one byte. Just remember that it does affect the D register and the C and Z bits.

APPENDIX 4 - INTERFACING A PRINTER

Any printer that will work with BASIC will work with SDS80C. Since there is sometimes a question about making a printer work with BASIC, however, this appendix is included to provide some information which may be helpful.

The Color Computer requires a serial printer. To connect a parallel printer, one needs a serial-to-parallel converter such as the PI80C which is sold by The Micro Works. In this case, merely plug the printer into the converter and the converter into the computer, leaving the baud rate at the default value of 600.

To run a serial printer, the "handshake" needs to be considered. The serial input line must be pulled up (to a "break" condition) in order for a printer to run. This is because the Output Character routine in the BASIC ROM checks this line after sending each character and waits for it to be high. This may be used to handshake with a printer if it provides a signal which is high when ready to accept a character. If this is not needed, the daring may install a 10K pullup resistor between this point (which may be found at the anode of CR6) and +5. (This resistor is large enough not to affect the operation of this line as an input.) If you do not want to modify your computer, you may need an external source of positive voltage.

The baud rate is set in locations $95 and $96. It defaults to 600 baud. It may be set in ABUG, by setting the bytes at locations $95 and $96 to the following values:

110 baud	01F3
300 baud	00B4
600 baud	0057
1200 baud	0029
2400 baud	0012

Note that except for 110 baud, the byte at $95 remains zero and need not be set.

If the cable is used which plugs into the serial port and has a DB-25 connector on the other end, the following points should be noted:

It is designed to plug into a modem. This means that it transmits on pin 2 of the 25-pin connector, and receives from pin 3. A printer will probably expect these two lines to be reversed; that is, the printer will listen on pin 3.

Ground is on pin 7 as usual.

The carrier-detect line goes to pin 8 and may be safely ignored.

An example of interfacing to a printer is given for the Malibu 165:

Receive Data is expected on pin 3 of the printer's DB-25 connector,

so this should be connected to the computer's Transmit line, or to pin 2 if the DB-25 cable is used.

A positive-true Printer Buffer Not Full signal is available on pin 20 of the printer's DB-25, and should be connected to the Receive line on the computer, or to pin 3 of the DB-25 cable.

If the Buffer Not Full signal is not used, the computer's Receive line should be pulled up and the baud rate should not exceed 600 (for this particular printer) so as not to overrun the printer's buffer.

If the printer used generates a carriage return automatically after column 32, use the "3" option when printing, as this will not send an extra carriage return. If your printer is wider than 32 columns, however, options "4" or "8" should be used.

Any printer connected to the Color Computer must generate a line feed after carriage return. This is a dipswitch option on many printers. The SDS80C will not send any line feed characters since this is the standard set by BASIC.

APPENDIX 5 - USE WITH THE DISASSEMBLER

The output of the Micro Works Disassembler is ordinarily sent to either the computer's display or to the RS232 output. It is possible, however, to direct the output to cassette tape. This is useful if one wishes to edit and reassemble the generated text.

Turn on the computer without SDS80C. This is so that BASIC can run.

Type "CLOADM" and load the disassembler, but do not run it.

Execute CBUG. If it is on ROM, it may be simply executed; if you have the tape version, offset load it to avoid interfering with the disassembler. Type:

```
CLOADM"CBUG",6666
EXEC
```

In CBUG, type "M 08F5" and enter the data "0A 6F 6F". This will set the output unit to -1 (tape) at the beginning of pass 2, as well as defaulting to high speed output (as with the "S" key).

Type "M 0973", and enter the data "7E A0 0E". This causes a return to BASIC when the disassembler is done.

Type "M 0887", and enter the data "12". This defaults to output format 4, which prints just the source text and no data columns. (The same thing can be accomplished by pressing the "4" key during pass 1.)

Type "G" and return to BASIC.

Start the output cassette and type:

```
OPEN"O",-1,"FILNAM"
```

This writes the file header.

Type EXEC 1540. This starts the disassembler.

Enter whatever parameters are desired for disassembly. When the disassembler is done, the screen will clear and say "OK".

Type "CLOSE" to write the end-of-file to the tape. The tape is done.

At this point, a power-off power-on reset is recommended, as after running any machine-language program such as the disassembler.

The tape may now be loaded by SDS80C. One warning: It is easy to generate a tape that is far to big to fit in memory. There is no point in trying to make a tape of the entire Basic ROM.

Variables and labels should now be changed with global changes into more meaningful names. Watch out for ORGs, and good luck.

APPENDIX 6 - ASCII CODE AND SCREEN CODE

Listed below is the ASCII code - the American Standard Code for Information Interchange. This is the code used by assembly language to represent text strings, and is the code which the assembler expects its source code to be in. Also listed below is the Color Computer screen code, which is the code in which the editor's text file is maintained while in RAM. The SDS80C does conversions between the two codes when entering and leaving the assembler.

ASCII	SCREEN	CHAR	ASCII	SCREEN	CHAR	ASCII	SCREEN	CHAR
20	60	space	40	40	@	60	--	'
21	61	!	41	41	A	61	01	a
22	62	"	42	42	B	62	02	b
23	63	#	43	43	C	63	03	c
24	64	$	44	44	D	64	04	d
25	65	%	45	45	E	65	05	e
26	66	&	46	46	F	66	06	f
27	67	'	47	47	G	67	07	g
28	68	(	48	48	H	68	08	h
29	69	)	49	49	I	69	09	i
2A	6A	*	4A	4A	J	6A	0A	j
2B	6B	+	4B	4B	K	6B	0B	k
2C	6C	,	4C	4C	L	6C	0C	l
2D	6D	-	4D	4D	M	6D	0D	m
2E	6E	.	4E	4E	N	6E	0E	n
2F	6F	/	4F	4F	O	6F	0F	o
30	70	0	50	50	P	70	10	p
31	71	1	51	51	Q	71	11	q
32	72	2	52	52	R	72	12	r
33	73	3	53	53	S	73	13	s
34	74	4	54	54	T	74	14	t
35	75	5	55	55	U	75	15	u
36	76	6	56	56	V	76	16	v
37	77	7	57	57	W	77	17	w
38	78	8	58	58	X	78	18	x
39	79	9	59	59	Y	79	19	y
3A	7A	:	5A	5A	Z	7A	1A	z
3B	7B	;	5B	5B	[	7B	--	{
3C	7C	<	5C	5C	\	7C	--	:
3D	7D	=	5D	5D	]	7D	--	}
3E	7E	>	5E	5E	^	7E	--	~
3F	7F	?	5F	5F	_	7F	--	rub

Note that on the screen of the Color Computer, the character "^" appears as an up-arrow, the character "_" appears as a back-arrow, and the lowercase letters appear as uppercase letters only inverted. Also note that carriage return (ENTER key) is $0D in ASCII. The editor in SDS80C uses null ($00) for this in the text buffer.

WARRANTY INFORMATION AND UPDATE REGISTRATION FORM

As with any software product, various improvements may be made from time to time. By sending in the form below, you will enable us to send you information about any future changes, so that you may have the option to send in your SDS80C to be updated to the newest version for a minimal handling charge.

Your SDS80C revision level is written on the label of the Rompack. Please mention this revision level when calling or writing to us with questions or comments about the SDS80C.

DO NOT OPEN the SDS80C Rompack. Doing so voids the warranty and makes you ineligible for the update service mentioned above. We cannot repair or replace Rompacks which have been tampered with.

The SDS80C Rompack is warranted to be free from defects in materials and workmanship for a period of one year from the date of purchase. The software is sold as-is without warranty. This warranty is in lieu of all other warranties, express or implied. This warranty does not apply to any Rompack which has been opened, or which has been modified or altered, or which has suffered abuse. Damage due to electrostatic discharge will be considered evidence of abuse.

If repairs or updates are required, and you ship via UPS, ship to our street address: 1942 S. El Camino Real, Encinitas, California 92024.

- -

UPDATE REGISTRATION PRODUCT: SDS80C

NAME: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

ADDRESS: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ ZIP _ _ _ _ _

REVISION LEVEL (See label of Rompack): _ _ _ _

DATE OF PURCHASE: _ _ _ _ _ _ _ _

PURCHASED FROM: (The Micro Works, or dealer name & city)

_ _

SAW ADVERTISEMENT IN:

_ _

Please clip this form and mail to The Micro Works, Inc., P O Box 1110, Del Mar, CA 92014.