COMPUTERWARE®
Microcomputer Sales and Software

Color BASIC
Compiler

[illegible], California 92024

The Color Compiler™
Version 2.0

Table of Contents

THE COLOR COMPILER™

PREFACE:

Computerware® is making a large investment in the software future of the Color Computer. We are working on software products at both the assembly and Basic Language level, as well as both serious and entertainment oriented. To achieve this goal, we need your support... One of the problems that developers of software have is that it takes a lot of initial time and money to 'create' the product before any revenue from its sale is generated. All too often when it is finished, customers who are not familiar with the development cycle for software products, see a cassette or disk and a manual and perceive that that is what the product cost. NOT TRUE!!

To be able to recover the development costs on inexpensive software, the manufacturer has to be able to sell a large number of copies. This is where you, the customer, can help by not giving away (or accepting from others) copyrighted software - actually any software product that is being offered for sale.

We have a lot of customers who tell us that they actively support us because they want our support in the years to come. When you think about that fact it makes sense. If we can't make enough sales because people are stealing copies of our products we will not continue to put our efforts into developing those products. So the bottom line is simply this: respect the copyright of software and do your part by not giving away or accepting copies of software that is offered for sale.

Thank You, Computerware®

LICENSE:

Computerware® Color Compiler™, in all machine readable formats, and the written documentation accompanying them are copyrighted. The purchase of Computerware® Color Compiler™ conveys to the purchaser a license to use Computerware® Color Compiler™ for his/her own use, and not for sale or free distribution to others. No other license, expressed or implied is granted.

WARRANTY INFORMATION:

The license to use Computerware® Color Compiler™ is sold AS IS without warranty. This warranty is in lieu of all other warranties expressed or implied. Computerware® does not warrant the suitability of the Color Compiler™ for any particular user application and will not be responsible for damages incidental to its use in a user system. If this product should fail to load during the first 90 days of use, simply return the ORIGINAL disk along with a copy of the receipt for a free replacement. After 90 days please innclude $8.00 to cover shipping and handling.

Computerware's Color BASIC Compiler V2.0
Program and Manual by Warren Ulrich III

If you have ever written a BASIC program only to find that it runs too slow to provide any action and haven't had the courage to learn assembler, then the Color Compiler™ is the answer to your problem. The Color Compiler™ lets you write your program in easy BASIC and then converts it into fast machine language. After you run your compiled program, you may find it necessary to add some delays because the Color Compiler™ will make your program run an average of 42 times faster. Some functions will run as much as 60 to 70 times faster!

The Color Compiler™ features a total of 55 instructions and functions. Most of these are a subset of Extended Color BASIC. Almost all of the graphics and sound functions are supported. This makes the Color Compiler™ ideal for writing graphics games and educational software which would run too slowly in Extended BASIC. Except for a few restrictions and non-implemented commands, you can program in BASIC and assume that ANYTHING is legal and the Color Compiler™ will understand. The Color Compiler™ was designed to run on a Color Computer with 32K of memory and at least one disk drive. The Color Compiler™ leaves approximately 16K of memory for your machine language program. The Color Compiler™ was made to be modular so instructions you may use frequently can be added to its vocabulary.

The Color Compiler™ generates position independent code so that you may put the compiled code anywhere in memory, including into a ROM-pack! It is extremely simple to pass variables back and forth between a BASIC driver program and the compiled program. Version 2.x allows string handling inside the compiled program to make your compiled code more versatile. A special feature of the Color Compiler™ allows machine language code to be embedded into the compiled program so you can do things that BASIC can't.

NOTE: Before you do anything else with your Color Compiler™ disk, MAKE A BACKUP COPY. This will save you a lot of time if you should accidentally delete a file or a whole disk. We ask that you respect the copyright that accompanies this software and not give away or sell copies. By doing this, we will be able to continue to provide good software at reasonable prices.

The subroutine package that is included in every compiled program is copyrighted. However, there is no additional fee to distribute a program written with the compiler. You must, however, include in the program and any documentation the words:

This program was created using the Color BASIC Compiler
(C)1984 Computerware

If you write a useful program with the compiler, please send a copy to Computerware for possible marketing.

SECTION I - HOW TO USE THE COLOR COMPILER:

1. SAVE the program to be compiled on any disk in any drive. DO NOT USE ASCII FORMAT. The Color Compiler™ reads the BASIC program directly from the disk and compiles it into memory. Make sure that the program uses only the instructions allowed by the Color Compiler™ and that it follows all the restrictions described in section II.

2. Put your Color Compiler™ diskette in drive 0 and type RUN"COMPILER". The Color Compiler will automatically execute a PCLEAR 0 to free up the maximum amount of memory possible. If the compiler does execute a PLEAR 0, it will reload itself. Be sure to PLEAR the number of graphics pages you will need after the compiler is done, otherwise you will get some unpredicable results when you execute the new program.

8000

3. Enter the address (in HEXIDECIMAL) where your machine code is to be stored. This will be the EXEC address of the resulting program (unless ROM is selected). NOTE: The Color Compiler™ uses hexidecimal for all numbers.

4. Enter the name of the program you saved in step 1. Make sure you enter the name the same way you saved it, even the drive number. The compiler will assume the file has the extension /BAS and that it is on drive 0 if you don't tell it something else. For example, if you enter 'MAZE', the compiler will assume you meant 'MAZE/BAS:0'.

5. Enter 'S' (or just press ENTER) for screen output or 'P' for printer output. The compiler will display the starting address of each BASIC line compiled. It will also display the CLEAR, START, END and EXEC addresses of the finished code. See section V for an explanation of these addresses.

6. Enter 'M' (or just press ENTER) for memory resident code or enter 'R' for code that is ROM-pack compatible. The ROMable code can be relocated into the &HC000 area and the arrays and strings (which normally exist just below the compiled program) will stay in RAM. The first three bytes of the ROMable code are a BRAnch to the beginning of your program so that when relocated to &HC000, a call to &HC000 will start the program (this is necessary for normal ROM-pack start up). Finally, the ROMable code has a JMP to BASIC's cold start routine at the end of the code. It is NOT intended to be used as subroutine for a BASIC program.

7. If you are passing values with the USR routine, enter a 'Y'. Otherwise, enter a 'N' (or just press ENTER).

SECTION I - HOW TO USE THE COLOR COMPILER (cont.)

8. The Color Compiler™ will now compile your program at the address given in step 3. This process may take a few minutes for longer programs. Just keep thinking how much faster it will run when the Color Compiler™ is done with it!

9. When the compiler is finished, if there were no errors, USR0 will be set to the starting address given in step 3. For more information on the USR function, see chapter 15 of your Extended BASIC manual.

SECTION II - RESTRICTIONS:

1. Maximum program length is 200 lines. This can be changed by setting the variable PL in line 0 to whatever value you need. If you try to make it too big, you will get an OM error (Out of Memory). Remember to re-SAVE the program if you make any changes that you want to keep.

2. Maximum number of line number references (GOTO's, GOSUB's, etc.) is 100. This can be changed by setting the variable LB in line 0 to whatever value you need. As with number 1 above, you are limited by memory available.

3. Variable names can only be single letter from A to Z and string variables from A$ to Z$.

4. All strings must be DIMensioned to the maximum length you will need. Strings can be from 0 to 255 characters.

5. Arrays can only be one dimension and single letter from A to Z. (These are seperate from simple variables). String arrays are NOT supported.

6. Only integers from -32768 to +32767 are allowed. The Color Compiler™ does not understand decimal numbers (like 3.1415, etc.).

7. All DATA statements must be the LAST statements (except for REMarks) in your program. All string DATA must be enclosed in quotes.

SECTION II - RESTRICTIONS (cont.)

8. All programs must have an END statement. The END statement may appear anywhere in your program except after DATA lines.

9. Only instructions listed in section IV and functions listed in section III are allowed. These must follow the syntax described in those sections. You will notice that some of the instructions syntax are different than that of BASIC's.

SECTION III - LEGEND FOR THE INSTRUCTIONS:

Section IV contains a list of all the instructions the Color Compiler™ understands. This legend should help you understand the syntax of each of these.

IE = an Integer Expression (Equation) that may have any combination of the following:

The arithmetic operators: + - * / ↑ ( ) < = >
The logical operators: AND OR NOT
Decimal Constants from -32768 to +32767
Hex Constants from &H0 to &HFFFF
Single letter variables A to Z
Array variables A(n) to Z(n)
String comparisons < = >
Any of the functions listed below:

- ABS(n): Returns the absolute value of n
- ASC(n$): Returns the ASCII code of n$
- JOYSTK(n): Returns the value of joystick n
- LEN(n$): Returns the length of n$
- PEEK(n): Returns the byte value at address n
- PEEK#(n): Returns the word value at address n
- PPOINT(x,y): Returns the color of the pixel at (x,y)
- RND(n): Returns a random number between 1 and n
- SGN(n): Returns the sign of n
- SQR(n): Returns the nearest integer square root
- TIMER: Returns the value of the timer
- VAL(n$): Returns the numeric value of n$

IC = Integer Constant

V = Simple Variable

AV = Array Variable

LN = Line Number

SECTION III - LEGEND FOR THE INSTRUCTIONS (cont.)

SE = String Expression (Equation) that may have any combination of the following:

String Addition: +
String constants contained within quotes
Single letter string variables A$ to Z$
Any of the functions listed below:
CHR$(n): Returns a character with the ASCII code n
INKEY$: Returns a character from the keyboard
MID$(n$,n,n): Returns a string portion (same as BASIC)
STR$(n): Returns the string equivalent of n

SC = String Constant. These must be enclosed in quotes. For example, "This is a string constant".

SV = String Variable

SECTION IV - INSTRUCTIONS ALLOWED:

With a few exceptions, all instructions have the same format and options as in Extended Color BASIC. Some exceptions are detailed with the commands. These modifications should not change programming but should in fact, make it easier.

R C HW S E
CIRCLE(IE,IE),IE,IE,IE,IE,IE

For the H/W ratio, use 0 to 1024, instead of 0 to 4, where 256 equals a perfect circle (128 would equal .5). For S and E (Starting and Ending points), use 0 to 64 instead of 0 to 1 (32 would equal .5). This change is needed since the Color Compiler™ does not understand decimals.

CLS IE

COLOR IE,IE

DATA IC,IC,"SC", ...
Note: All DATA must come at the end of your program. All string DATA must be enclosed in quotes. String constants and numeric constants may be mixed on the same line, but should be separated for clarity.

DIM AV(IC),AV(IC),SV(IC) ...
Note: Only Integer Constants may be used.
String variables must be DIMensioned for the maximum length they will contain (0 to 255).

END
Note: Every program must have an END statement.
The END statement tells the compiler where you want to jump back to BASIC.

EXEC IE
Note: Exec causes a JSR to the address specified by IE.

FOR V = IE TO IE STEP IE

GET(IE,IE)-(IE,IE),AV (All options are supported)

GOSUB LN

GOTO LN

IF IE THEN LN ELSE LN
Note: Only line Numbers may be used.

LET and Implied Let:
V = IE or AV = IE or TIMER = IE or SV = SE

LINE(IE,IE)-(IE,IE),PSET (All options are supported)

MOTOR ON or OFF

NEXT V,V, ... (All options are supported)

ON IE GOSUB LN,LN, ...

ON IE GOTO LN,LN,...

PAINT(IE,IE),IE,IE

PCLS IE

PCOPY IE TO IE

PMODE IE,IE

POKE IE,IE (Pokes single bytes)

POKE# IE,IE (Pokes two byte words)

PRESET(IE,IE)

PRINT IE,IE, or PRINT @ IE, or PRINT SE;IE , etc.
Note: Any combination is valid.

PSET(IE,IE,IE)

PUT(IE,IE)-(IE,IE),AV (All options are supported)

READ V,V,AV,SV, ...
Note: Any combination is valid.

REM or '

RESTORE

RETURN

SCREEN IE,IE

SOUND IE,IE

USR;HC;HC;HC... (HC is a hex constant)

This instruction allows the user to add machine language instructions within the compiled program. All of the numbers must be in hexidecimal and must be seperated by semicolons. As an example:

```
10 USR;83;0;0;30;1F;26;FC;86;4F;BD;A2;82
```

Represents:

```
830000        LDX  #$0000
301F    WAIT  LEAX -1,X
26FC        BNE  WAIT
864F        LDA  #$4F
BDA282      JSR  >$A282
```

This command will be useful to assembly language programmers who want to embed machine code directly into their compiled programs.

SECTION V - ADDITIONAL INFORMATION:

1. Passing simple variables from BASIC:

When you set up a program to be compiled and you wish to pass a variable from a BASIC driver program, simply leave that variable undefined at the BEGINNING of the program to be compiled. EXAMPLE: B=A*24 where the variable A has not been defined before in the compiled program. You should set all variables used by the compiled program (not passed from BASIC) to zero at the beginning of the program.

2. Passing the USR value from BASIC:

After answering 'Y' to the question in step 6 in section I, the variable 'U' will automatically contain the USR value at the start of your compiled program.

3. Passing the USR value back to BASIC:

After answering 'Y' to the question in step 6 in section I, assign the variable 'U' equal to the value you wish to return. This will be passed back to the calling program.

4. Obtaining the remainder from a divide:

By using the % sign IMMEDIATELY after a divide, you can obtain the remainder. EXAMPLE: A=B/C+% or A=B/C:B=%

5. What the CLEAR, START, END, and EXEC addresses are:

When the Color Compiler™ is finished working on your program, it will display four addresses. You can save the compiled program to disk by typing:

```
SAVEM"FILENAME",&H(START),&H(END),&H(EXEC)
```

where START, END and EXEC are the addresses printed by the compiler. Before executing the program, you should CLEAR memory with:

```
CLEAR (STRING SPACE),&H(CLEAR)
```

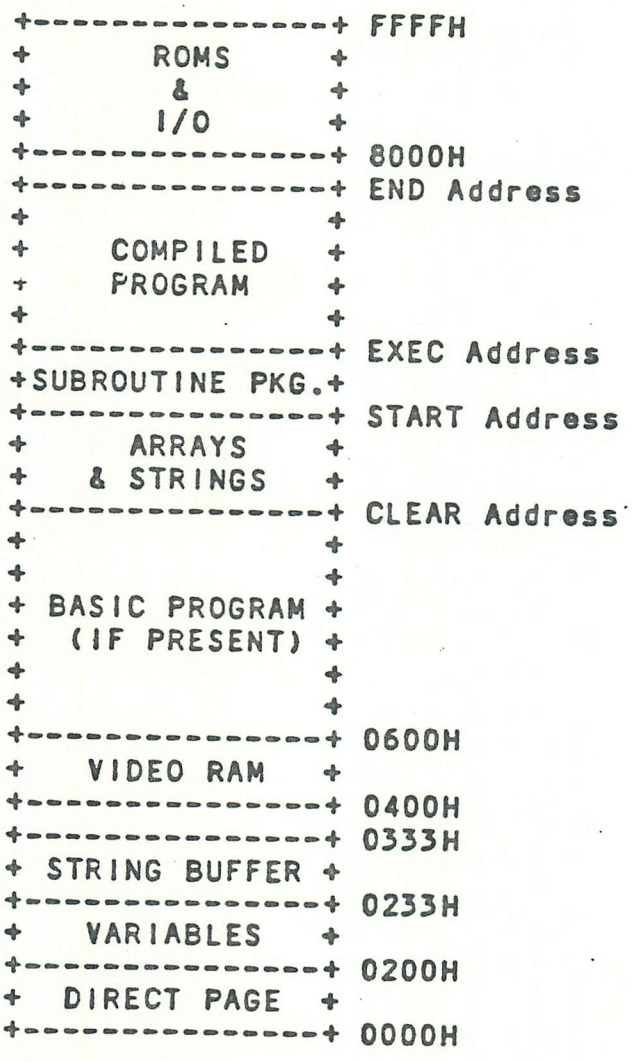
where CLEAR is the CLEAR address printed by the compiler and STRING SPACE is the amount of memory to reserve for strings used by BASIC. This will insure the proper location of the stack pointer. Also, since the compiled code cannot execute the proper PCLEAR statement, you must be sure that either you or the calling BASIC program does so. If you don't, you will get unpredictable results ranging from a ?FC ERROR to a crashed computer.

SECTION V - ADDITIONAL INFORMATION (Cont.)

6. All arrays have random values after the DIM statements and should therefor be cleared at the beginning of the program. Strings are all set to null.

7. All machine code generated by the Color Compiler™ is completely relocatable. WARNING: The compiler generates code that uses some of the routines in EXTENDED COLOR BASIC. Therefore, most compiled programs will not work on a non-extended computer.

8. The following is a memory map showing how your program is compiled into memory:



SECTION VI - TIPS & TRICKS:

1. To calculate the array sizes for GET and PUT instructions:

   H = rectangle height W = rectangle width
   For PMODE 0 : ARRAY SIZE = H*W/32+4
   1 : ARRAY SIZE = H*W/16+4
   2 : ARRAY SIZE = H*W/16+4
   3 : ARRAY SIZE = H*W/8+4
   4 : ARRAY SIZE = H*W/8+4
   H and W are figured using standard 256 X 192 coordinates, and not by counting the number of actual picture elements.

2. Do not be afraid of using GOSUB's or ON n GOSUB/GOTO's. These instructions will save lots of memory and are extremely fast.

3. The SQR function is designed to handle unsigned amounts from 0 to 65535 (0 to FFFFH) and give the closest integer result.

4. If you wish to make good sound effects, use 0 (zero) as the length in the SOUND instruction. Color BASIC does not accept a zero but the compiler will. By using zero as the length, a very short duration is sounded. By mixing different frequencies, you can make a lot of different sounds.

5. If you need to add some delays to your program, try using the TIMER instead of a FOR/NEXT loop. It will give you more predictable results and smoother animation. The timer increments sixty times per second so you can get accurate delays from a sixtieth of a second to many minutes. For example, to get a X second delay call this subroutine:

```
1000 REM DELAY FOR X SECONDS
1010 TIMER=0
1020 IF TIMER<X*60 THEN 1020
1030 RETURN
```

6. You may have many subroutines compiled at once by putting them all into the same program to be compiled followed with an END. While the compiler is working, write down the address of the first line of each subroutine. You may simply EXEC to these addresses to access any part of the program. Be careful not to EXEC into the middle of a FOR-NEXT loop in the compiled program.

SECTION VI - TIPS & TRICKS (cont.)

7. The following subroutine will input a string into A$ with length L:

```
10 DIM A$(255),I$(1)
      .
      .
      .
1000 A$="":L=32
1010 I$=INKEY$:IF I$="" THEN 1010
1020 IF I$=CHR$(13) THEN 1100
1030 IF I$=CHR$(8) THEN 1080
1040 IF ASC(I$)<32 THEN 1010
1050 IF LEN(A$)=L THEN 1010
1060 A$=A$+I$
1070 PRINT I$;:GOTO 1010
1080 IF LEN(A$)=0 THEN 1010
1090 A$=MID$(A$,1,LEN(A$)-1):GOTO 1070
1100 PRINT:RETURN
```

SECTION VII - ERROR MESSAGES:

If the Color Compiler™ cannot compile a line of the source program, it will stop and print an error message. These are similar to BASIC's errors, however you can tell them apart because the compiler does not precede the error message with a question mark. Below is a list of possible error messages and their likely causes:

ERROR	POSSIBLE CAUSE
DD	Double DImensioned array. -An array variable was DIMensioned more than once. -A string variable was DIMensioned more than once.
ME	Missing End -The END statement was left out of the program. -All programs must have an END statement.
NE	Name does not Exist. -The program name you gave the compiler does not exist on the disk. Check the DIRectory.
OS	Out of Space. -The program compiled beyond the 7FFFH limit. -There were too many program lines. -There were too many line number references. -A string became larger than 255 characters.
SN	Syntax. -A typical typing error. -A decimal point in a constant. -An illegal instruction. -An instruction (other than REMarks) after DATA. -An instruction instead of a line number in an IF/THEN statement. -A two letter variable. -An array with more than one dimension.
TM	Type Mismatch -String and numeric were mixed in the same formula.
UA	Undefined Array. -All arrays must be DIMensioned even if you are using 10 or less cells.
UL	Undefined Line number reference. -The number given in this case is not the line where the error occurred, but the line number the Color Compiler™ cannot find.
US	Undefined String. -All strings must have a DIMension length.

SECTION VIII - SAMPLE RUNS:

The following program was run with each of the different line 30's and timed. The timings for both the compiled and BASIC versions are listed below.

```
10 PMODE4,1:A=88:B=20:TIMER=0
20 FOR N=1 TO 10000
30 *
40 NEXT N:U=TIMER:END
```

LINE 30	COMPILED(sec.)	BASIC(sec.)	SPEED DIFF.
* REM	.68	32.27	47:1
* C=A	.82	42.68	52:1
* C=A+B	1.07	55.67	52:1
* C=A-B	1.32	57.10	43:1
* C=A*B	2.83	58.33	21:1
* C=A/B	6.57	88.57	13:1
* C=ABS(A)	1.05	51.97	49:1
* C=JOYSTK(0)	5.00	102.58	21:1
* C=PEEK(N)	.95	59.33	62:1
* C=PPOINT(A,B)	3.23	75.65	23:1
* C=RND(A)	9.42	131.72	14:1
* C=SGN(A)	1.00	55.07	55:1
* C=SQR(A)	9.58	631.07	66:1
* C=TIMER	.82	45.62	56:1
* GOSUB50/50RETURN	.85	47.93	56:1
* IFA<B THEN40	1.18	58.53	50:1
* IFA>B THEN40	1.22	61.60	50:1
* POKE A,B	1.02	54.55	53:1
* PRESET(A,B)	3.80	63.15	17:1
* PSET(A,B)	3.80	61.66	16:1
* RESTORE:READA	1.48	105.20	71:1
	AVERAGE SPEED INCREASE IS		42:1

SECTION IX - USING THE DEMO PROGRAMS:

Your Color Compiler™ disk contains the following files:

```
COMPILER.BAS   <-  The packed version of the Compiler™
COMPILER.REM   <-  The REMarked version of the Compiler™
SUBPACKG.BIN   <-  The subroutine package
SUBPACKG.TXT   <-  The subroutine package source code
MAZE    .BAS   <-  A demo program
BEAM    .BAS   <-  A demo program
FUNSOUND.BAS   <-  A demo program (NOT to be compiled)
BEEP    .BAS   <-  Used by FUNSOUND
BEEP    .BIN   <-  Compiled version of above
COUNTER .BAS   <-  A demo program
FUNPLOT .BAS   <-  A demo program
DUMP    .BAS   <-  A demo program
```

SUBPACKG.BIN and SUBPACKG.TXT are the subroutine package that is added to the beginning of every compiled program. This is a position independent file and must remain that way if you make any changes to it.

MAZE.BAS is an excellent example of the speed difference created by the Color Compiler™. Compile the program at &H7800 and then type 'CLEAR 200,&H6680:A=USR(0)'. This program runs about 40 times faster once it has been compiled.

BEAM.BAS is a game that resembles the Tron light cycles. It was written only to be compiled (you can't run the source code). Compile it at &H7700. The object is to surround your opponent with your trail and make him crash into a wall. The joystick button controls the speed of your cycle.

FUNSOUND.BAS calls BEEP.BIN to demonstrate the sound function with a duration of 0. You can change the variable I to get different sorts of sound effects. This also shows the USR value being passed to the compiled program.

COUNTER.BAS is a subroutine that displays numbers on the PMODE 1 screen. It is set up to rapidly display the numbers 1 to 500, but can be easily modified to listen to a BASIC driver program.

FUNPLOT.BAS plots JOYSTK(0) vs. time with sound. This was the first program we wrote to go with our Bio-Detector™. It shows how fast you can do graphics with a compiled program.

DUMP.BAS dumps the ASCII equivalent of a range of memory to the screen. At the 'START:' prompt, enter the first address to dump (in hex). Enter the last address to dump at the 'END :' prompt. While the screen is scrolling, press any key to pause it. Any key (other than 'Q') will resume the listing. If you press 'Q', the program will start over. If you enter '0000' for both the start and end addresses, the program will exit to BASIC.

SECTION X - TECHNICAL INFORMATION:

The following few pages are provided for advanced programmers who may want to modify the Color Compiler™ to add additional commands or change the way existing ones work. Keep in mind that once you have modified the program, Computerware cannot help you with any problems that may arise.

A. SUBROUTINES:

The following is a list of the major subroutines used by the Color Compiler™. This will be helpful if you intend to add additional instructions. NOTE: The line numbers here are for the REMarked version of the program. We suggest that you make any changes to this version and after testing, remove the REMarks and renumber the modified version. Make sure that, if you do make any changes and save the program, you change the name in line 1430 (RUN"COMPILER/REM") to match the name you used to save the new version. Otherwise, when the compiler has to reload itself after a PCLEAR 0, it will load the old version.

GET NEXT CHARACTER LINE# 60

ENTRY CONDITIONS: none.
EXIT CONDITIONS: Next character is returned in C. CC points to the current character.

GET PREVIOUS CHARACTER LINE# 80

ENTRY CONDITIONS: none.
EXIT CONDITIONS: CC points to the previous character. GOSUB 60 to obtain C. WARNING: This routine is only for obtaining the last character again. GOSUB 60 must be called between GOSUB 80's.

DECIMAL NUMBER DECODE LINE# 130

ENTRY CONDITIONS: CC points to the first digit.
EXIT CONDITIONS: N contains the value. CC points to the last digit.

HEX NUMBER DECODE LINE# 150

ENTRY CONDITIONS: CC points to the first digit of the number to decode (not &H).
EXIT CONDITIONS: N contains the value. CC points to the last digit.

A. SUBROUTINES (cont.)

POKE BYTE LINE# 180

ENTRY CONDITIONS: P contains the value to poke.
EXIT CONDITIONS: M is incremented by one to the next byte.

POKE WORD LINE#190

ENTRY CONDITIONS: P contains the 16 bit value to poke.
EXIT CONDITIONS: M is incremented by 2 to the next word.

FIND VARIABLE ADDRESS LINE# 250

ENTRY CONDITIONS: V contains the ASCII code minus 65. W0 contains 0 for simple variables, 1 for arrays, or 2 for strings.
EXIT CONDITIONS: V contains the address or base address for arrays and strings.

EXPRESSION DECODE LINE# 330,340

ENTRY CONDITIONS: CC points to the character just before the expression. GOSUB 330 for numeric expressions. GOSUB340 for string expressions. TM error check is at 1390 for strings 1400 for numeric. The error is automatically checked if 330 and 340 are called.
EXIT CONDITIONS: The expression value will be calculated into the D register for numeric. For strings the X register points to the beginning and the B register contains the length. CC points to the next character after the expression.

B. VARIABLE DEFINITIONS:

VARIABLE	MEANING
A	Miscellaneous use.
AA	Current array address pointer.
AD	Memory poke address (double byte).
C	Next character in ASCII.
CC	Current character pointer.
CG	Current granule number.
CS	Current sector number.
CV	Current variable address.
DE	Output device number.
DF	Data flag.
DR	Disk drive number.
E	End statement flag.
EF	End of program flag.
FL	Disk read flag.
GP	Line number table pointer.
LB	Maximum number of line references.
LM	Don't skip spaces flag.
LN	Current line number.
LP	Line reference table pointer.
[illegible]	Top of line number table address.
M	Current memory poke address.
MF	Lowest save address.
MS	Execute address.
N	Constant value/Error trap value.
[illegible]	Next granule.
NS	Next sector flag.
OK	Error type flag.
P	Poke value.
PL	Maximum number of program lines.
RF	ROM-Pack flag.
SF	String flag/expression type.
SP	Length of the Subroutine Package.
TP	Line number table pointer.
V	Variable name (ASCII-65).
W0	Var. type/ 0=simple,1=array,2=string.
W-W9	Working storage.

ARRAY VARIABLE DEFINITIONS:

A(n)	Line number reference table.
AA(n)	Array variable address table.
LT(n)	Line number table.
SA(n)	String variable address table.
VA(n)	Simple variable address table.

STRING VARIABLE DEFINITIONS:

A$	First half of sector contents.
B$	Second half of sector contents.
EX$	Extension name.
F$	Program name.
M$	USR value flag.
N$	Number constant.

C. SUBROUTINE PACKAGE POINTERS:

VARIABLE	SUBROUTINE AND CONDITIONS
AC	DESCRIPTION: Points to the array subscript check routine. This routine calculates the array pointer, checks its value against the dimensioned size, and leaves the pointer on the top of the stack. ENTRY CONDITIONS: The array base address must be the first value on the stack (besides the return address). The subscript value must be in D. EXIT CONDITIONS: The pointer to the array value is left on the top of the stack. To obtain the array value use the following instruction: LDD [,S++] or equiv.
CO	DESCRIPTION: Prints spaces to the next comma field. ENTRY CONDITIONS: none EXIT CONDITIONS: none
DA	DESCRIPTION: Divide routine. Divides two signed 16 bit values. ENTRY CONDITIONS: The dividend (first value) must be in X. The divisor (second value) must be in D. EXIT CONDITIONS: The quotient is returned in D, and the remainder is stored in addresses $5C/$5D.
DA	DESCRIPTION: Read data routine. Reads one 16 bit data item into D and increments the pointer. ENTRY CONDITIONS: none. EXIT CONDITIONS: Returns a value in D.
ET	DESCRIPTION: Prints 'ENTER' ASCII Code ENTRY CONDITIONS: none EXIT CONDITIONS: none
IK	DESCRIPTION: Scans the keyboard for a pressed key. This routine is the same as INKEY$ in BASIC. ENTRY CONDITIONS: none EXIT CONDITIONS: The X register points to the string buffer where the character is stored and the B register contains the length (0 or 1).
JA	DESCRIPTION: Read joystick routine. Reads one joystick value at a time instead of all four like Color BASIC. ENTRY CONDITIONS: D should contain the value of which joystick to read. EXIT CONDITIONS: Returns the joystick value in address $51.

C. SUBROUTINE PACKAGE POINTERS (Cont.):

VARIABLE — SUBROUTINE AND CONDITIONS

MA — DESCRIPTION: Multiply routine. Multiplies two signed 16 bit values.
ENTRY CONDITIONS: The first value must be in X. The second value must be in D.
EXIT CONDITIONS: The value of the multiply is returned in D.

MI — DESCRIPTION: MID$ function. This routine returns a portion of a string. ENTRY CONDITIONS: The stack must be in the following order:

LOW memory: Return address for the subroutine jump.
Return string length. FFFFH = balance.
Starting point in string.
String total length.
HIGH memory: String address.

EXIT CONDITIONS: The X register points to the string portion. B equals the new string length.

NA — DESCRIPTION: Points to the NEXT routine. This routine increments the variable given in the FOR instruction by the STEP value and checks it against the limit. If the value is outside of the limit, the routine jumps out of the FOR/NEXT loop.
ENTRY CONDITIONS: The stack must be in the following order:

LOW memory: Return address from the subroutine jump.
STEP value (16 bit).
Jump address to the start of the loop.
Limit value (16 bit).
HIGH memory: Pointer to the FOR/NEXT variable.

EXIT CONDITIONS: none

NO — DESCRIPTION: Prints a number on the screen.
ENTRY CONDITIONS: Number to print must be in D.
EXIT CONDITIONS: none

PA — DESCRIPTION: PPOINT routine. Gets the color at the X/Y point on the screen (same as Extended Color BASIC).
ENTRY CONDITIONS: The X coordinate must be stored at $BD/$BE. The Y coordinate must be stored at $BF/$C0.
EXIT CONDITIONS: The point color value is returned in D.

C. SUBROUTINE PACKAGE POINTERS (Cont.):

VARIABLE — SUBROUTINE AND CONDITIONS

RA — DESCRIPTION: Random number generator. This routine calculates a random number, using the TIMER, between one and the value of the argument.
ENTRY CONDITIONS: The maximum value must be in D.
EXIT CONDITIONS: A random number is returned in D.

RS — DESCRIPTION: Read String routine. This routine reads a string into a variable and increments the pointer to the next data item.
ENTRY CONDITIONS: The U register must contain the pointer to the variable.
EXIT CONDITIONS: The variable will automatically contain the string data only if there was no size conflict.

SA — DESCRIPTION: Square root routine. This routine calculates the square root for any number (unsigned) between 0 and 65535 (0 to FFFF Hex).
ENTRY CONDITIONS: The number to find the square root of must be in D.
EXIT CONDITIONS: The nearest integer square root is returned in D.

SB — DESCRIPTION: String append routine. This routine addes one string to another creating one long string.
ENTRY CONDITIONS: The first string's pointer and length must be on the stack. The second string's pointer must be in X and its length in B.
EXIT CONDITIONS: The X register contains the pointer to the beginning of the string buffer and the B register contains the string's length.

SC — DESCRIPTION: String compare routine. This routine compares two strings and returns a flag that indicates the result of the comparison.
ENTRY CONDITIONS: The first string's pointer and length must be on the stack. The second string's pointer must be in X and its length in B.
EXIT CONDITIONS: The B register contains a flag and the CC register refects the contents of B. The following is a list of what the flags mean:

B	Meaning
$FF	The 1st string < 2nd string.
0	The 1st string = 2nd string.
1	The 1st string > 2nd string.

SD — DESCRIPTION: STR$ function. This routine changes a 16 bit integer into an ASCII string.
ENTRY CONDITIONS: The D register must contain the number.
EXIT CONDITIONS: The X register points to the beginning of the string and B contains the length.

C. SUBROUTINE PACKAGE POINTERS (cont.)

VARIABLE	SUBROUTINE AND CONDITIONS
SO	DESCRIPTION: String output routine. This routine outputs a string of characters. ENTRY CONDITIONS: The X register must contain a pointer to the beginning of the string. The B register contains the string length. EXIT CONDITIONS: none
ST	DESCRIPTION: String transfer routine. This routine transfers a string of characters into a string variable. ENTRY CONDITIONS: The U register must point to the string variable. The X and B registers must contain the pointer and length of the string to transfer. EXIT CONDITIONS: Registers U, X, and B are all modified.
VA	DESCRIPTION: VAL function. This routine returns the signed numerical value of a string. ENTRY CONDITIONS: The X register must point to the beginning of the string and the B register must contain its length. EXIT CONDITIONS: The D register contains the signed integer value of the string.

D. HOW THE COLOR COMPILER WORKS:

In order to fully understand how the Color Compiler™ works, you will have to understand how BASIC works and have some knowledge of machine language programming. Here is a quick overview of what BASIC does when you type in a program line:

As soon as you press ENTER, BASIC changes your commands and functions into codes called tokens. The tokens and the information that makes up the rest of the line, including a code for the line number and a two byte offset to the next line, are stored in the program at the appropriate spot. When you SAVE your program to disk, BASIC does not change this format (UNLESS you use ASCII format!).

When you compile your program, the Color Compiler™ finds the location on the disk where your program is stored and reads the information directly. Here is what happens:

- First, the Color Compiler™ reads the line offset value, which is generally ignored.
- Second, it saves the line number and the location in memory where it starts for an update routine later in the program.
- Third, the Color Compiler™ gets an instruction, in token form, and decodes it according to its syntax.
- Fourth, after decoding the instruction and poking the machine language equivalent into memory, the compiler™ looks for a code, which is the end of the line, or the code for a colon which means more instructions. If it is a zero the compiler returns to the first step. If the code is a colon, the compiler returns to the third step.
- Fifth, the process continues until the end of program flag is found (a zero for the offset code).
- Sixth, the Color Compiler™ now updates all the jumps (GOTO's and GOSUB's) that were accumulated in the first pass.
- Last, the compiler prints all the important information about the compiled program and sets USR0 to the start address.

Here is an example of how a program line looks on the disk:

Your program line: 10 READ A(N)

What BASIC actually stores in memory: 27 FF 00 0A 8D 20 41 28 4E 29 00
--A-- --B-- C D E F G H I

A is the offset to the next line.
B is the line number.
C is the token for READ.
D is the ASCII code for a space.
E is the ASCII code for the A.
F is the ASCII code for the (.
G is the ASCII code for the N.
H is the ASCII code for the).
I is the end of line flag.

E. HOW TO ADD YOUR OWN INSTRUCTIONS:

If you have little or no knowledge of how to program in machine language, this section will be hard to understand. This information is provided for experienced programmers only.

If you are going to add an instruction or function to the Color Compiler's vocabulary, it must already be in BASIC's vocabulary. If it isn't, there will be no token generated by BASIC and the compiler will see it as a variable. As an example, let's add the AUDIO ON/OFF instructions.

Add the following lines to the compiler program:

```
1885 IFC=&HA1 GOSUB5000  'AUDIO
5000 GOSUB60
5010 IFC=&H88 THENP=&H5F:GOSUB180:W=&HA99D:GOSUB1280:GOTO60
5020 IFC<>&HAA THEN2100
5030 W=&HA974:GOSUB1280:GOTO60
```

Line # 1885 tells the compiler that the decoding routine for AUDIO is at line 5000.

Line # 5000 gets the next character after the AUDIO token.

Line # 5010 checks that character and if it is the token for ON (&H88) it pokes the code for a CLRB (&H5F), pokes a JSR to the AUDIO ON routine in the BASIC ROM (&HA99D), then gets the next character and returns to the main loop.

Line # 5020 checks that character and if it is not the token for OFF it reports a SN (syntax) error.

Line # 5030 pokes the JSR to the AUDIO OFF routine in the BASIC ROM (&HA974) and then gets the next character and returns to the main loop.

Keep in mind that in this case most of the work was done in BASIC's ROM. On the other hand, most additions that use more parameters or equations won't be as simple unless you are good at programming in machine language. All of the main subroutines and some commonly used machine language instructions are between lines 60 and 1410. These will help you cut down on adding a lot of poke lines (GOSUB180 & 190's). Functions can be added in the same manor as instructions between lines 720 and 880. See the next page for a list of instruction and function tokens. Here is what the compiler changes POKE A+1,20 into:

```
LDD     >$200      The $200 is Variable A's location.
PSHS    A,B        Save D on the stack.
LDD     #1         Constant 1.
ADDD    ,S++       Add 1 to A.
PSHS    A,B        Save poke address.
LDD     #$14       $14 is HEX for 20.
STB     [,S++]     Stores B at the address on the stack
                   and strips the stack.
```

INSTRUCTION TOKENS:

'	83	DIR	CE	LOAD	D3	RENAME	D6
*	AD	DLOAD	CA	LSET	D4	RENUM	CB
+	AB	DRAW	C6	MERGE	D5	RESET	9D
-	AC	DRIVE	CF	MOTOR	9F	RESTORE	8F
/	AE	DSKI$	DF	NEW	96	RETURN	90
<	B4	DSKINI	DC	NEXT	8B	RSET	D7
=	B3	DSKO$	E0	NOT	A8	RUN	8E
>	B2	EDIT	B6	OFF	AA	SAVE	D8
AND	B0	ELSE	84	ON	88	SCREEN	BF
AUDIO	A1	END	8A	OPEN	99	SET	9C
BACKUP	DD	EXEC	A2	OR	B1	SKIPF	A3
CIRCLE	C2	FIELD	D0	PAINT	C3	SOUND	A0
CLEAR	95	FILES	D1	PCLEAR	C0	STEP	A9
CLOAD	97	FN	CC	PCLS	BC	STOP	91
CLOSE	9A	FOR	80	PCOPY	C7	SUB	A6
CLS	9E	GET	C4	PLAY	C9	TAB(	A4
COLOR	C1	GO	81	PMODE	C8	THEN	A7
CONT	93	IF	85	POKE	92	TO	A5
COPY	DE	INPUT	89	PRESET	BE	TROFF	B8
CSAVE	98	KILL	D2	PRINT	87	TRON	B7
DATA	86	LET	BA	PSET	BD	UNLOAD	DB
DEF	B9	LINE	BB	PUT	C5	USING	CD
DEL	B5	LIST	94	READ	8D	VERIFY	DA
DIM	8C	LLIST	9B	REM	82	WRITE	D9
●	AF						

FUNCTION TOKENS:

ABS	82	LOG	99
ASC	8A	MEM	93
ATN	94	MID$	90
CHR$	8B	MKN$	A6
COS	95	PEEK	86
CVN	A2	POINT	91
EOF	8C	POS	9A
EXP	97	PPOINT	A0
FIX	98	RIGHT$	8F
FREE	A3	RND	84
HEX$	9C	SGN	80
INKEY$	92	SIN	85
INSTR	9E	STRING$	A1
INT	81	STR$	88
JOYSTK	8D	SQR	9B
LEFT$	8E	TAN	96
LEN	87	TIMER	9F
LOC	A4	USR	83
LOF	A5	VAL	89
		VARPTR	9D