COMPUTERWARE®

Specialists in Application
and System Software

BASIC Compiler

We Sell Capabilities . . .

4403 Manchester Ave. Suite 102-Box 668
Encinitas, Calif. 92024 (619) 436-3512

COMPUTERWARE COLOR COMPILER™
Version 2.4

Table of Contents

SECTION TITLE PAGE

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:

The Computerware® Color Compiler™, in all machine readable formats, and the written documentation accompanying them are copyrighted. The purchase of the Computerware® Color Compiler™ conveys to the purchaser a license to use the 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 the 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 include $8.00 to cover shipping and handling.

Computerware®'s Color BASIC Compiler V2.4
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 30 to 40 times faster. Some functions will run as much as 60 to 70 times faster!

TO GET THE MOST OUT OF YOUR COLOR COMPILER, READ THROUGH THIS MANUAL COMPLETELY...

The Color Compiler™ features a total of 69 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.

Version 2.4 has the following new features:

1) New instructions and functions:

- CLEAR
- DRAW
- LINE INPUT
- MID$() =
- PCLEAR
- PLAY
- HEX$
- INSTR
- LEFT$
- RIGHT$
- STRING$

New Features contd....

2) Enhanced Instructions and fuctions:
   - IF ;handles all combinations
   - DIM ;allows scalars (String and Numeric)
   - PRINT ;allows PRINT# and TAB(n)
   - READ ;gives an ?OD ERROR if out of data
   - VAL ;can handle 16 bit signed and unsigned values

3) Other new features:
   - Numeric & string arrays can be multi-dimensional
   - TRACE ;helps debug programs
   - Loads Source program on compile error
   - Allows for Relocatable or Compact object code
   - Printer Pagination
   - Warning for possible undefined simple variables
   - Expanded more powerful subroutine package
   - Uses about the same amount of memory as version 2.1

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 quality software at reasonable prices.

The subroutine packawe that is included in every compiled program is copyrighted. 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:

Parts of this program were created using the 'Color BASIC Compiler' (C)1985 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:

Once you have created a BASIC source program and have checked to be sure that it obeys all the restrictions described in section II and that all the commands follow the syntax described in sections III & IV, you are ready to compile it. Follow the instructions below to complete this process.

1. SAVE the program to be compiled (your SOURCE code) 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.

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 PCLEAR 0, it will reload itself. Be sure not to remove the program disk from drive 0 until the program is finished. Be sure to PCLEAR the number of graphics pages you will need.

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). The Color Compiler™ uses hexidecimal for all numbers.

   NOTE: You do not have to type in the &H before the memory location you wish to start at. If you enter an address that is too low or too high, you will get a 'BAD ADDRESS' message. For most programs, using 7000 gives you a good starting point.

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 unless you 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 and instruction compiled. It will also display the CLEAR, START, END and EXEC addresses of the finished OBJECT code and the addresses of all the variables. See section V for an explanation of these addresses.

   Be sure to align the printer paper at this time.

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, strings, and variables (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 a subroutine for a BASIC program.

NOTE: Options 7,8 and 9 are not available if ROM is selected.

7. If you are having trouble with your object program, press 'Y' to activate the trace function or 'N' (or just press ENTER) to skip. The TRACE routine adds 48 bytes to the beginning of your program plus six bytes for each line of source code. Be sure to compensate for this so you don't run out of memory.

   How to operate the trace:
   - A. Compile the program.
   - B. CLEAR memory and type EXEC.
   - C. Hold the 'R' key down to speed through the program, lift to pause.
   - D. Press the 'S' key to step line by line.
   - E. Press the BREAK key to terminate the program.
   - F. The numbers in brackets represent the line number of the source program just before it executes.
   - G. DO NOT use CONT after a BREAK or ERROR.

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

9. Press 'R' (or just press ENTER) to make the object code relocatable or press 'C' to make it more compact and execute quicker. If 'C' is selected, the object code will not be relocatable.

10. 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!

11. When the compiler is finished, and there were no errors, the compiler will print the CLEAR, START, END, and EXEC addresses then ask you for the name of the new program. If you enter a name, the compiled program will be saved on the disk. If you just press 'ENTER', the program will not be saved.

12. At this point, the compiler will print all the variable addresses and set USR0 and EXEC to the execution address. For more information on the USR function, see chapter 15 of your Extended BASIC manual.

SECTION II - RESTRICTIONS:

The Color BASIC Compiler has a number of limitations that Extended Color BASIC does not have. You will probably find that these won't hinder your programming. In fact, some may even make programming easier. A list of restrictions follows.

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. All strings must be DIMensioned to the maximum length you will need. Strings can be from 0 to 255 characters. See section IV under DIM for dimensioning strings and string arrays.

4. Only integers from -32768 to +32769 are allowed. The Color Compiler™ does not understand decimal numbers (like 3.1415, etc.).

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

6. 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 Co Compiler" understands. This legend should help you underst the syntax of each of these.

Following most of the instructions and functions is a numb and letter in brackets like this: [141S]. The number stands the page in either the Standard BASIC manual (S) or the Exten BASIC manual (X) where the instruction or function is found. most cases, the Color Compiler will use the same format options described in the manuals. If no number appears, instruction or function is not supported by COLOR BASIC.

IE = an Integer Expression (Equation) that may have 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
- Any length variable names (2 char. significant)
- Multi-dimensional Array variables
- String comparisons < = >
- Any of the functions listed below:

```
Numeric Functions:
------------------
ABS(n): Absolute value  [300S] *
ASC(n$): ASCII code [300S] *
INSTR(n,s$,n$): Search string [118X] *
INT(n): Acts as a parenthesis [302S] *
JOYSTK(n): Joystick value [302S] &
LEN(n$): String length [303S] *
PEEK(n): Byte value [304S] *
PEEK#(n): Returns the 16 bit value at address n
PPOINT(x,y): Point color [29X] *
RND(n): Random number [305S] *
SGN(n): Sign [305S] *
SQR(n): Square Root [102X] *
TIMER: Timer value [140X] *
VAL(n$): String value [305S] *

* Same as Color BASIC.
& Reads one joystick value at a time.
```

IC = Integer Constant

V = Simple Variable

AV = Array Variable

LN = Line Number

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

- String Addition: +
- String constants contained within quotes
- Any length string variable names (2 char. significant)
- Multi-dimensional string arrays
- Any of the functions listed below:

```
String Functions:
-----------------
CHR$(n): String Character [300S] *
HEX$(n): Hexidecimal value [142X] *
INKEY$: Scan Keyboard [302S] *
LEFT$(n$,n): Left string [302S] *
MID$(n$,n,n): Mid-string [303S] *
RIGHT$(n$,n): Right string [305S] *
STR$(n): Numeric to string [305S] *
STRING$(n,n$): String of characters [117X] *

* Same as Color BASIC.
```

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

SV = String Variable

SA = String Array

SECTION IV - INSTRUCTIONS ALLOWED:

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

```
          R  C  HW S  E
CIRCLE(IE,IE),IE,IE,IE,IE,IE [41X]
```

For the H/W ratio, use 0 to 1024,linstead 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.

```
CLEAR IE,IE [300S]
```

CLS IE [301S]

COLOR IE,IE [14X]

DATA IC,IC,"SC", ... [301S]

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

DIM AV(IC,IC,IC),SA(IC,IC,IC),SV,SV(IC),V, ... [301S]

Note: Only Integer Constants may be used. String variable must be DIMensioned for the maximum length they wil contain (0 to 255). Example: DIM SV$(15) wil dimension a string variable for a maximum of 1 characters. If a string variable (SV) appears b itself, the compiler will assume its length to be 12 characters. If a numeric variable (V) appears b itself, the compiler will allocate space for th variable (2 bytes) and clear that space apon executio of the DIM statement in the compiled code. It i recommended that all variables be defined in this wa at the beginning of your program.

String arrays are DIMensioned as follows:

10 DIM AA$(c,e,e) 'c' is the maximum number o characters per string (0 to 255). 'e' is the numbe of elements per dimension.

DRAW SE [53X]

NOTE: All options are supported except the X funtion and th undocumented = function. If you do have an X or function in the DRAW string, the compiler will use th variables from BASIC's variable table and not th compiled variables.

END [301S]

NOTE: The END statement tells the compiler where you want t jump back to BASIC.

EXEC IE [301S]

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

FOR V = IE TO IE STEP IE [302S]

GET(IE,IE)-(IE,IE),AV (All options are supported) [67X]

GETV,V=n,AV=n,SV=s,SA=s, ...

Note: This command allows you to pass BASIC variables to compiled variables. The n and s stand for the BASIC (n)umeric and (s)tring variables to be passed. Arrays may also be passed.

As an example, let's pass BASIC's array BB$ to the compiled array AA$:

```
Source program:
10 DIM AA$(30,10,10)  'DIM A$ to 30 char, 10 by 10 element
20 FOR XX=0 TO 10  'Set up X loop
30 FOR YY=0 TO 10  'Set up Y loop
40 GETV,AA$(XX,YY)=BB$(XX,YY)  'Transfer variable
50 NEXT YY,XX  'End of loops
60 END 'Back to BASIC
```

Note that compiled variables always are on the left side of the equal sign and BASIC's variables are always on the right. Think of it as V <= n.

GOSUB LN [302S]

GOTO LN [302S]

IF IE THEN ... ELSE ... [302S]

Note: All combinations are supported (even nested IF's).

LET and Implied Let [17S] [139X]

Note: Any and all combinations are supported.

LINE(IE,IE)-(IE,IE),PSET (All options are supported) [9X]

LINEINPUT ON "PROMT";SV [127X]

NOTE: The ON is an option that allows you to activate the BREAK key. If you leave the ON option out, the BREAK key will act the same as the ENTER key. If the ON is included, the program will be terminated when the BREAK key is pressed.

MID$(SV,IE,IE) = (Same as BASIC) [122X]

MOTOR ON or OFF [303S]

NEXT V,V, ... (All options are supported) [302S]

ON IE GOSUB LN,LN, ... [303S]

ON IE GOTO LN,LN, ... [303S]

PAINT(IE,IE),IE,IE (All options are supported) [49X]

PCLEAR IE [26X]

PCLS IE [25X]

PCOPY IE TO IE [28X]

PLAY SE [73X]
NOTE: All options are supported except the X and
functions. See DRAW for more details.

PMODE IE,IE (All options are supported) [19X]

POKE IE,IE (Pokes single bytes) [304S]

POKE# IE,IE (Pokes two byte 16 bit words)

PRESET(IE,IE) [6X]

PRINT IE,IE or PRINT @ IE or PRINT SE,IE or PRINT #IE etc..[304S
Note: Any and all combinations are valid. TAB(n) is als
supported. The USING function is not supported at th
time.

PSET(IE,IE,IE) [3X]

PUT(IE,IE)-(IE,IE),AV (All options are supported) [67X]

PUTV,V=n,SV=s,AV=n,SA=s, ...
Note: This instruction works the same as GETV except th
the compiled variable is passed back to the BAS
variable. Remember that the variable on the left si
of the equal sign is always the compiled variable a
the variable on the right is always the BAS
variable. Think of it as V => n. See GETV for mo
details.

READ V,V,AV,SV,SA ... [304S]
Note: Any combination is valid.

REM or ' [304S]

RESTORE [304S]

RETURN [304S]

SCREEN IE,IE [35X]

SOUND IE,IE [305S]

Note: A duration of 0 is allowed and can be used to create some very interesting sounds.

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

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

```
10 USR;27;7;C6;1;4D;2A;1;50;1D;39
```

Represents:

```
*********
*SGN FUNCTION (from the Subroutine Package)
*********
2707              BEQ   ZIP
C601              LDB   #1
4D                TSTA
2A01              BPL   POSITV
50                NEGB
1D       POSITV   SEX
39       ZIP      RTS
```

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 variables to and from BASIC:

To pass variables, whether they are numeric, string, or arrays of either kind, use the GETV and PUTV instructions described in section IV. Any combination, i.e. numeric variable to numeric array, etc., is valid. However, there are two restrictions:

1. You cannot mix variables (string and numeric).
2. If you attempt to pass a string from BASIC with GETV whose length exceeds the compiled string's DIMensioned length, you will get an ?CS ERROR.

Additional Info contd...

2. Passing the USR value from BASIC:

After answering 'Y' to the question in step 8 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 8 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. The START, END, and EXEC addresses are used to save the program to disk. The CLEAR address is the lowest memory used for variable storage by the compiled program. 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 that BASIC doesn't overwrite your compiled program or its variable table. Also, if your new program uses graphics, be sure it PCLEARs enough graphics pages. If you don't, you will get unpredictable results ranging from a ?FC ERROR to a crashed computer.

6. All arrays are filled with zeros by the DIM statements. Strings and string arrays are all set to null.

7. All machine code generated by the Color Compiler™ is completely relocatable (unless the Compact option is selected). 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:

```
+---------------+ FFFFH
+     ROMS      +
+      &        +
+     I/O       +
+---------------+ 8000H
+---------------+ END Address
+               +
+   COMPILED    +
+   PROGRAM     +
+               +
+---------------+ EXEC Address
+SUBROUTINE PKG.+
+---------------+ START Address (ROM EXEC addr.)
+  VARIABLES,   +
+    ARRAYS     +
+  & STRINGS    +
+---------------+ CLEAR Address
+               +
+               +
+ BASIC PROGRAM +
+ (IF  PRESENT) +
+               +
+               +
+---------------+ 0600H
+   VIDEO RAM   +
+---------------+ 0400H
+---------------+ 03FFH
+ STRING BUFFER +
+---------------+ 0200H
+---------------+ 0100H
+  DIRECT PAGE  +
+---------------+ 0000H
```

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. To obtain the fastest results from GET and PUT, use arrays that have only one dimension and calculate the size using the formulas above. Arrays used for GET and PUT with more than one dimension will give slower results.

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 ELSE RETURN
```

   To syncronize your graphics with the video screen call this routine:

```
        2000 REM VIDEO SYNC. ROUTINE
        2010 TIMER=0
        2020 IF TIMER THEN RETURN ELSE 2020
```

6. You may have many subroutines compiled at once by putting them all into the same program to be compiled followed with a RETURN. 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.

7. The JOYSTK function in Color Basic is handled somewhat differently than the way the Color Compiler does. In Color Basic, you are required to find the value of JOYSTK(0) first before finding the value of 1,2, or 3. JOYSTK(0), in this case, reads all the joystick values at once. The Color Compiler, however, only reads the joystick value you ask for. This way you'll get much quicker results because you're not looking at the other joystick values you don't need.

SECTION VII - ERROR MESSAGES (COMPILE-TIME):

If the Color Compiler™ cannot compile a line of the source program, it will stop, print an error message, and load the source program. These error messages are similar to BASIC's errors, but 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. -A string array was DIMensioned more than once.
EF	ELSE without an IF -There were more ELSE's than IF's in the same line.
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. -You were out of variable memory space. -A string became larger than 255 characters.
OV	Overflow -A numeric contant went beyond -32768 to 32767.
PUV	Possible Undefined Variable -The compiler has found a variable in the current line that may be undefined. This error is only a warning and will not cause the compiler to terminate.
SN	Syntax. -A typical typing error. -Instruction format difference. -A decimal point in a constant. -An illegal instruction. -An illegal function. -An instruction (other than REMarks) after DATA. -Something other than variables in GETV and PUTV.
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 VII - ERROR MESSAGES (RUN-TIME):

Run-time error checking was kept to a minimum to get the maximum program speed possible. However, the following errors will occur when variable data or the compiled program are in danger of being changed. When these errors occur, they will resemble standard BASIC errors. In this case, however, the line number (if shown) refers to the BASIC line where the jump was made to the compiled program. If you are running the TRACE routine, the line number that appears after the error refers to the line in your source code.

ERROR

POSSIBLE CAUSE

?BS

Bad Subscript
-An array subscript value went beyond the DIMensioned limit.
-The number of subscripts in the array did't match up with the number DIMensioned.

?FC

Function Call
-Not enough graphics pages PCLEARed.
-MID$ starting point was 0.
-INSTR starting point was 0.
-The array was not large enough for GET or PUT.

?OD

Out cf Data
-A READ command went beyond the last data item.

?OS

Out cf String space
-A string was larger than the variable's DIMensioned length.
-Two strings added together became larger than 255 characters.

?SN

Syntax
-The string used in MID$()= was not a variable. In this case, a string or string array variable are the only type allowed.

?ST

String expression Too complex
-The string expression (equation) became too complex to handle. Break it up into smaller expressions.

SECTION VIII - SAMPLE RUNS:

The following program was run with each of the different line 50's and timed. The timings for both the compiled and BASIC versions are listed below.

```
10 DIM TEST$(20),T2$(20),TA(36)
20 PMODE4,1:GET(0,0)-(15,15),TA
30 TEST$="SEE HOW FAST I AM!":A=88:B=20:TIMER=0
40 FOR N=1 TO 10000
50 *
60 NEXT N:PRINT TIMER:END
```

LINE 50	COMPILED(sec.)	BASIC(sec.)	SPEED DIFF.
* REM	.68	32.27	47:1
* C=A	.92	42.68	46:1
* C=A+B	1.20	55.67	46:1
* C=A-B	1.50	57.10	38:1
* C=A*B	3.38	58.33	17:1
* C=A/B	7.17	88.57	12:1
* C=ABS(A)	1.15	51.97	45:1
* C=JOYSTK(0)	5.18	102.58	20:1
* C=PEEK(N)	1.32	59.33	45:1
* C=PPOINT(A,B)	3.30	75.65	23:1
* C=RND(A)	8.75	131.72	15:1
* C=SGN(A)	1.20	55.07	46:1
* C=SQR(A)	9.72	631.07	65:1
* T2$=TEST$	5.40	48.65	9:1
* GOSUB 70/70 RETURN	.83	47.93	58:1
* IFA<B THEN60	1.40	58.53	42:1
* IFA>B THEN60	1.63	61.60	38:1
* POKE A,B	1.10	54.55	50:1
* PUT(0,0)-(15,15),TA	37.0	L66.98	5:1
* PSET(A,B)	3.88	61.66	16:1
* RESTORE:READA	1.67	129.27	77:1

AVERAGE SPEED INCREASE IS 36:1

In all of the compiled timings, the relocatable option was used instead of the compact option. If the compact option is selected, the speed will increase on the average of 10%. You will notice that while some commands are speeded up significantly, others (PUT in particular) do not gain much from being compiled. This is because Extended BASIC already handles these efficiently. Remember, each one of these commands is executed 10,000 times!

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
SORTER /BAS	<-	Calls SORTSUB/BIN (don't compile)
SORTSUB /BAS	<-	An array sort subroutine
DIRSUB /BAS	<-	A subroutine to sort your directory
SORTDIR /BAS	<-	Calls DIRSUB (don't compile)

SUBPACKG.BIN and SUBPACKG.TXT are the subroutine package that i added to the beginning of every compiled program. This is position independent file and must remain that way if you mak any changes to it. If you do make any changes, be sure th length of the package is the same as the variable SP is set to i line 40. Also, make sure all the pointers set in lines 1700 an 1710 are in alignment with each of the routines in the package.

NOTE: When answering the prompts from the compiler, all of th demos should be compiled using the compiler's defaults. Thi means (S)creen display, (M)emory, (N)o trace, (N)o USR value, an (R)elocatible.

MAZE.BAS is an excellent example of the speed difference create by the Color Compiler™. Run the BASIC version first (it take approx. 45 min. to finish!). Now, compile the program at 760 and then type 'EXEC'. This program runs about 45 times faste once it has been compiled.

BEAM.BAS is a game that resembles the Tron light cycles. It wa written only to be compiled (you can't run the source code) Compile it at &H7600 then type 'EXEC'. The object is to surroun your opponent with your trail and make him crash into a wall The joystick button controls the speed of your cycle.

SORTER.BAS calls the compiled SORTSUB.BAS to sort array SORT$(n in alphabetical order (quickly!). NE contains the number o elements to sort and array SORT$(n) is replaced with the sorte elements. This is an excellent example of the GETV and PUT commands. DO NOT COMPILE SORTER.BAS!! SORTSUB.BAS should b compiled at &H7D00.

SORTDIR.BAS calls the compiled DIRSUB.BAS (DIRSUB.BIN) and wil sort your disk directory into alphabetical order. DIRSUB.BA should be compiled at &H7A00.

SECTION X - TECHNICAL INFORMATION:

The following 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 RENUM 0,,1 the modified version. Make sure that, if you do make any changes and save the program, you change the name in line 4190 (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. Also, if you add any new variables, make sure you define them in the DIM in line 1730.

GET NEXT CHARACTER LINE# 60

ENTRY CONDITIONS: Set variable LM to 0 to automatically skip spaces. Set LM to 1 to accept spaces as valid characters.
EXIT CONDITIONS: Next character is returned in C in ASCII form. 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 (continued)

POKE BYTE LINE# 190

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

POKE WORD LINE# 200

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: C contains the ASCII code of the first character in the variable name.
EXIT CONDITIONS: VA contains the address for variables or base address for arrays and strings. W contains the variable name + 32768 for strings + 128 for arrays WO contains:

0 = Simple numeric
1 = Numeric array
2 = Simple string
3 = String Array

EXPRESSION DECODE LINES# 360,370

ENTRY CONDITIONS: CC points to the character just before the expression. GOSUB360 for numeric expressions. GOSUB370 for string expressions. TM error check is at 1560 for strings 1570 for numeric. The error is automatically checked if 360 and 370 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 of the string and the B register contains its length. CC points to the next character after the expression. C will contain the character just after the expression.

334

B. VARIABLE DEFINITIONS:

VARIABLE	MEANING
AA	Current variable address pointer.
AD	Memory poke address (double byte).
C	Next character in ASCII.
CC	Current character pointer.
CG	Current granule number.
CM	Compact/relocatable flag.
CS	Current sector number.
CV	Current variable table pointer.
DE	Output device number.
DF	Data flag.
DR	Disk drive number.
E	End statement flag.
EF	End of program flag.
EP	ELSE table pointer.
ES	Ending sector.
FL	Disk read flag.
FP	IF table pointer.
G	GOTO flag.
GP	Line number table pointer.
LB	Maximum number of line references.
LC	Output line counter.
LM	Don't skip spaces flag.
LN	Current line number.
LP	Line reference table pointer.
LS	Top of line number table address.
M	Current memory poke address.
ME	Program end address.
MF	Lowest save address.
MS	Execute address.
N	Constant value/Error trap value.
NG	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.
TR	Trace flag.
V	Variable table base address.
VA	Returned variable address.
WO	Var. type/ 0=simple,1=array,2=string,3=string array.
W-W9	Working storage.
Z	Store address/misc.

ARRAY VARIABLE DEFINITIONS:

A(n)	Line number reference table.
EA(n)	ELSE address table.
FC(n)	IF address table.
LT(n)	Line number table.

STRING VARIABLE DEFINITIONS:

A$	First half of sector contents.
B$	Second half of sector contents.
K$	Program name.
M$	USR value flag.
N$	Number constant/misc.

C. SUBROUTINE PACKAGE POINTERS:

VARIABLE SUBROUTINE AND CONDITIONS

AB

DESCRIPTION: Points to the string array subscript check routine. This routine calculates the array pointer, checks its value against the dimensioned size, and leaves the pointer on the stack.
ENTRY CONDITIONS: Same as AC below.
EXIT CONDITIONS: The pointer to the array value is left on the top of the stack. To obtain the array pointer use the following instructon : LDX ,S++ or equiv.

AC

DESCRIPTION: Points to the numeric array subscript check routine. This routine calculates the array pointer, checks the subscipt values against their dimensioned sizes, and leaves the pointer on the top of the stack.
ENTRY CONDITIONS: The stack must be in the following order:

```
LOW  memory: Subroutine RTS
             Subscript N (16bit)
                  :
                  :
             Subscript 2
             Subscript 1
HIGH memory: Array base address
```

The B register must contain the number of subscripts on the stack.
EXIT CONDITIONS: Same as AB above. Use LDD [,S++] or equiv. to obtain the array value.

AF

DESCRIPTION: AND, OR, and NOT operators.
ENTRY CONDITIONS: For AND and OR the first value must be on the stack and the second value in the D register. For NOT the value should be in the D register.
EXIT CONDITIONS: The result of all is returned in the D register.

CH DESCRIPTION: CHR$ function. Creates an ASCII character.
ENTRY CONDITIONS: The B register must contain the ASCII code of the character.
EXIT CONDITIONS: The X register points to the character and B contains a 1.

CI DESCRIPTION: CIRCLE command.

CP DESCRIPTION: Numeric compare subroutine. Compares two numeric value according to the call (<=>) and leaves either a true value (FFFFH) or a false value (0) in the D register.

DA DESCRIPTION: Divide routine. Divides two signed 16 bit values.
ENTRY CONDITIONS: The dividend (first value) must be on the stack. The divisor (second value) must be in D.
EXIT CONDITIONS: The quotient is returned in D, and the remainder is stored in addresses $77/$78.

DI DESCRIPTION: DIMension routine. This routine sets up all arrays at run-time.
ENTRY CONDITIONS: The U register contains the ending address of the array. The X register points to the beginning. The B register contains the number of subscripts on the stack. The stack must be in the following order:

```
LOW  memory: Subroutine RTS (16bit)
             Subscript N (16bit)
                  :
                  :
             Subscript 2 (16bit)
             Subscript 1 (16bit)
HIGH memory: String length (8bit) (zero for numeric)
```

DS 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.

DW DESCRIPTION: DRAW command.

EC DESCRIPTION: END command. Gets the original stack address of BASIC and returns to BASIC.

GT DESCRIPTION: GET command.

GV DESCRIPTION: Gets a value from a BASIC variable and stores it in a compiled variable.
ENTRY CONDITIONS: The A register must be cleared. The U register points to the compiled variable. X contains the name of the BASIC variable. Any array subscripts should be stored in reverse order from address $400 to lower memory. Call the routine as follows:

CALL	BASIC variable type
GV - 3	for a string array
GV - 2	for a string variable
GV - 1	for a numeric array
GV	for a numeric variable

EXIT CONDITIONS: None. The variable is automatically stored at the address pointed to by U.

HX DESCRIPTION: HEX$ function. This routine takes the value in the D register and converts it into a string that represents the hexidecimal value.

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).
IN°DESCRIPTION: INSTR function. Same as Extended BASIC.

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 the D register.

LE DESCRIPTION: LINE command.

LI DESCRIPTION: LINE INPUT command.

MA

DESCRIPTION: Multiply routine. Multiplies two signed 16 bit values.
ENTRY CONDITIONS: The first value must be on the stack. 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.

MT

DESCRIPTION: MID$()= command. This command replaces a portion of a string with the result of the string expression.

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

NE

DESCRIPTION: This short routine negates the contents of D.
ENTRY CONDITIONS: The value to negate must be in D.
EXIT CONDITIONS: The sign of D is the opposite of what it was.

NO

DESCRIPTION: Prints a number on the screen.
ENTRY CONDITIONS: Number to print must be in D.
EXIT CONDITIONS: none

OG DESCRIPTION: ON GOTO/GOSUB commands.

OP DESCRIPTION: OPEN STACK SPACE for array subscripts. This routine saves the value of the D register (the last subscript value) on the stack, increments the number of subscripts, and opens up two more bytes on the stack for the next subscript.

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 color value is returned in D.

PK DESCRIPTION: PEEK function.

PM DESCRIPTION: PMODE command.

PT DESCRIPTION: PAINT command.

PV DESCRIPTION: This routine takes a value from a compiled variable and puts it in a BASIC variable.
ENTRY CONDITIONS: Same as GV except use PV through PV - 3 to call the routine.
EXIT CONDITIONS: none. The value of the variable is automatically stored in BASIC's table.

PY DESCRIPTION: PLAY command.

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. The random number seed is at address $FA.

RI DESCRIPTION: RIGHT$ function.

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.

RT DESCRIPTION: RESTORE command.

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 !ts length in B.
EXIT CONDITIONS: The A register contains a flag and the CC register refects the contents of A. The following is a list of what the flags mean:

A	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.

SG DESCRIPTION: SGN function.

SI DESCRIPTION: STRING$ function.

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

SS DESCRIPTION: PUSH STRING. Anytime a string must be saved on the stack for later, this routine should be called. If not, and the string is in the string buffer, there is a possibility of it being over written.

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.

VL 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 this 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 zero, 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, saves it to disk, and sets the USR0 pointer to the start address. The EXEC address is also automatically set to the start address when the subroutine package is linked.

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

```
our program line:     10 READ A(N)

hat BASIC actually
tores 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. (10 in HEX)
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.

. HOW TO ADD YOUR OWN INSTRUCTIONS:

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

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

dd the following lines to the REMARKED compiler program:

```
2055 IFW=161THEN5000   'AUDIO
5000 IFC=136GOSUB1210:W=&HA99D:GOSUB1360:GOTO60
5010 IFC<>170THEN2280
5020 W=&HA974:GOSUB1360:GOTO60
```

Line # 2055 tells the compiler that the decoding routine for AUDIO (token 161) is at line 5000.

Line # 5000 checks the next character and if it is the token for ON (token 136) it pokes the code for a CLRB (GOSUB1210), pokes a JSR to the AUDIO ON routine in the BASIC ROM (&HA99D), then gets the next character and returns to the main loop.

If C was not the token for ON, line # 5010 checks the character and if it is not the token for OFF (token 170) it reports a SN (syntax) error.

Line # 5020 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 1590. These will help you cut down on adding a lot of poke lines (GOSUB190 & 200's). Functions can be added in the same manor as instructions between lines 800 and 1040. See the next page for a list of instruction and function tokens. Here is what the compiler changes POKE A+1,20 into:

```
     LDD      >VA,PCR    The VA is Variable A's location (relative).
     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.
```

To get a better idea of what the compiler is doing, let's follow the SOUND instruction through its decoding routine.

```
2050 IFW=160THEN2990 'SOUND
       :
       :
2980 REM **SOUND**
2990 GOSUB350:P=55180:GOSUB200:GOSUB1540:GOSUB360:W=43345:GOTO1360
        A     -------B--------       C          D       -------E--------
```

[1]. The token for SOUND is 160 which is found at line 2050. [2]. Line 2050 tells the compiler that the decode routine for SOUND is on line 2990. [3]. First SOUND calls the expression decode routine (A) to get the value for the frequency. Note that it did'nt call line 360. The reason for this is because line 350 skips getting the first character in the expression (we already have it; see the GOSUB60 in line 1890). [4]. Next, it pokes the value 55180 (B) into memory (this saves the frequency value in a location required by Color BASIC).

[5]. Next, GOSUB1540 (C) checks to make sure the next character is a comma. [6]. Next, the routine calls the expression decode routine again (D) to get the length of the SOUND command (This time we called 360). [7]. Last, the routine pokes a JSR (E) to the SOUND routine in BASIC ROM. [8]. One last note. In this case, the expression decoding routine at line 360 returned with the character just after the expression. In some other cases, the next character may not be returned. Apon returning to the main loop, it is required that the next character be in C (and it must be a zero or colon). Make sure your routines do this or you will keep getting a SN error when you compile a program with the new instruction in it.

F. 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								

F. FUNCTION TOKENS (preceeded by $FF):

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