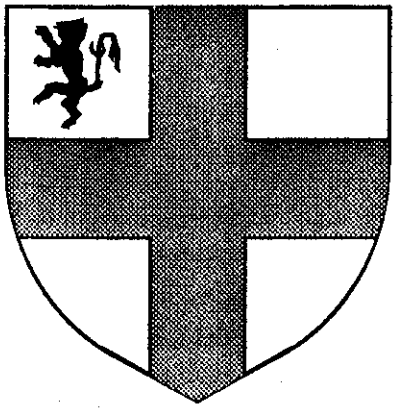
R . S . B .

Disk BASIC for Level 2 OS9
Version 1.02



P.O. Box 58342
Renton, WA 98058

PUBLISHED BY BURKE & BURKE
P.O. BOX 1283
PALATINE, IL 60078-1283

COPYRIGHT NOTICE

LIMITED WARRANTY

Burke & Burke warrants the PRODUCT

against defects in material or workmanship for a period of ninety (90) days from the date of purchase by the original owner. This warranty is limited to repair or replacement of PRODUCT which proves to be defective during this period, at the sole expense and discretion of Burke & Burke. This warranty specifically excludes defects in software and defects caused by abuse, negligence, accident, or tampering.

DISCLAIMER

Every effort has been made to ensure the accuracy of this manual and the quality of the PRODUCT it describes. Burke & Burke makes no warranties, whether expressed, statutory, or implied, of any kind whatsoever, as to the merchantability of the PRODUCT or it's fitness for a particular use, except as set forth above as the LIMITED WARRANTY.

Burke & Burke does not assume any liability for any damages, whether special, indirect, or consequential, resulting from the use of the PRODUCT. The PRODUCT is sold on an AS-IS basis.

TRADEMARKS

Disk Extended Color BASIC is a copyrighted software product of Microsoft and Microware Systems Corporation, licensed to Tandy Corporation. Color Computer and CoCo are trademarks of Tandy Corporation.

Contents

Chapter 1

Introduction

R.S.B. is a full OS9 adaptation of Disk Extended Color BASIC (DECB) for OS9 Level 2 running on the Color Computer 3.

Color Computer users will benefit from RSB's familiar command syntax and excellent compatibility with existing BASIC software. Advanced OS9 programmers will find that RSB is an excellent tool for rapid development of portable software.

RSB is a version of Disk Extended Color BASIC that has been modified for compatibility with Level 2 OS9. Key features of RSB include:

* Fully re-entrant; BASIC prorams may be run in several windows simultaneously.

* Fully relocatable.

* Command syntax is identical to Disk Extended Color BASIC.

* Accepts commands in either upper or lower case.

* All I/O, including disk, graphics and sound, uses OS9 system calls.

* VDG graphics commands work on both VDG screens and Level 2 windows.

* High-resolution graphics.

* Tandy Color Mouse or Deluxe Color Mouse can be used in place of joysticks.

* Multi-Vue compatible.

* New commands give BASIC programs direct access to OS9.

This manual describes how to use RSB, and provides information about how OS9 and RSB interact. Nonetheless, the manual does not attempt to teach BASIC programming or the use of OS9. If you are not familiar with programming in Disk Extended Color BASIC, refer to Tandy's Color Computer 3 Extended BASIC and Color Computer Disk System reference manuals.

Required Equipment

The following equipment is required by RSB:

* Tandy Color Computer 3 (128K minimum)
* Television or Monitor
* Disk Drive
* OS9 Level 2 Operating System
* Floppy disk controller with one of the following ROM's installed:

    a) Tandy Disk Extended Color BASIC 1.0
    or
    b) Tandy Disk Extended Color BASIC 1.1
    or
    c) Tandy Disk Extended Color BASIC 2.0
    or

d) Tandy Disk Extended Color BASIC 2.1
or
e) DISTO CoCo 3 CDOS Disk BASIC

The following equipment is optional. Using some or all of this equipment will enhance the operation of RSB:

* Printer
* Second disk drive
* Joystick(s) or mouse
* Speech/Sound PAK or Super Voice
* RS-232 PAK
* Hard disk system
* Multi-Vue environment (512K)
* 512K RAM

Although RSB will run in 128K of RAM, Burke & Burke strongly recommends that you use 512K for best performance.

Installation

There are two steps to installing RSB on your Color Computer 3:

1) Run the INSTALL procedure to create an executable copy of RSB.

2) Edit the RSB environment file to tailor RSB to your hardware.

RSB is distributed on two diskettes: the INSTALLATION DISK, and the DEMO / UTILITIES DISK. Neither disk is copy protected. Before starting to install RSB, you should make a backup copy of each disk and store the originals in a safe place. USE THE ORIGINAL RSB DISKS ONLY TO CREATE A BACKUP

COPY.

The RSB distibution disks don't include an executable copy of RSB. Instead, the INSTALL procedure file builds an executable copy of RSB from the data in your Color Computer's BASIC and DISK BASIC ROMs. Once you have an executable copy of RSB, you can run RSB directly (without repeating the installation procedure).

It takes all of your Color Computer's resources to read the ROMs when INSTALL is running, so be sure that you don't have any applications running in other windows. You must also run INSTALL from a true window, not a 32 column VDG window. If you don't normally use true windows, you can create one by entering the commands:

```
OS9:shell i=/w7& [ENTER]
&xxx
OS9:[CLEAR]
```

You should see a screen with the word Shell and the OS9: prompt in the upper left-hand corner.

To create an executable copy of RSB, from a true window, with no other programs running, insert a BACKUP COPY of the RSB INSTALLATION disk in drive 0 and enter:

```
OS9:chd /d0
OS9:chx /d0
OS9:install
RSB Installation in Progress . . .
Modifying BASIC 2.X . . .
Installation complete.
```

You will see a lot of disk activity, and

may see a flashing black bar at the bottom of your screen (the flashing bar indicates that the installation program is accessing your Color Computer's ROMs).

The INSTALL program takes about 10 minutes to run. When it finishes, there will be an executable copy of RSB in the root directory of drive /d0.

The RSB program is stored in a file called RSB. You can copy RSB to a working system disk, or can run the program right from the installation disk.

To copy the RSB program to a system disk, place the system disk in drive 1 and enter the commands:

```
OS9:copy /d0/rsb /d1/cmds/rsb
OS9:copy /d0/sys/rsb_env.file /d1/sys/
    rsb_env.file
```

Be sure to use a system disk with enough room on it (you need at least 260 free clusters), and be sure to create the SYS directory on the system disk if it isn't there already.

RSB is supplied configured for use with typical Color Computer 3 systems. If you have a hard disk, mouse, or other advanced hardware, you may want to edit the RSB environment file so that RSB will access these devices. See Appendix C for further information on the RSB environment file.

Using RSB

RSB is an OS9 command program. You can

run RSB from the OS9 Shell, Multi-Vue, a Shell procedure file, or another program. The command line used to start up RSB looks like this:

```
RSB options program_name #memory_size
```

All of the arguments are optional. If no arguments are specified, RSB comes up in Direct Mode with about 5K of RAM available for program and data storage.

The options argument lets you enable or disable certain features of RSB (such as VDG graphics).

The program name argument specifies the OS9 path to a file that you want RSB to load and execute automatically. If the file does not exist, RSB generates an error message and comes up in Direct Mode.

The memory size argument tells the OS9 Shell how much RAM to give RSB for program and variable storage. The default value of this argument is 8K. Whatever argument is supplied, RSB uses the first 3k of RAM for its internal variables.

Available Options

Options are specified on RSB's command line using the prefix '-'. The available options are:

-g Disables allocation of a VDG graphics screen at RSB startup. Specifying this option saves 6K of RAM, but also disables graphics on VDG windows. Note

that PMODE graphics will still work on true windows even when the -g option is specified.

EXAMPLES:

rsb -g #20K

Does not allocate a VDG graphics screen but allocates 20K of RAM to RSB. About 17K of this RAM will be available for program and variable storage.

rsb /d0/basic/payroll.bas

Tells RSB to run the payroll.bas program, which is in the directory /d0/basic.

Chapter 2

General Information

RSB is an interpreter for a dialect of the BASIC computer programming language. The specific dialect, Disk Extended Color BASIC (DECB), was originally developed for stand-alone use in the Tandy Color Computer 3.

Most DECB programs run under RSB with no changes. Throughout this manual, the term BASIC is used to refer to either RSB or Disk Extended Color BASIC.

Direct vs Run Mode

You are always in one of two operating modes when using RSB: Direct Mode (program editing, command execution) or Run Mode (BASIC program execution).

Direct Mode lets you enter commands that load, save, create, modify, erase, or run programs. Typical direct mode commands are:

```
load "quack.bas"
list 100-220
tron
copy "test.dat" to "test.bak"
del 9000-
run
```


RSB is in Direct Mode whenever it is not executing a program.

In Run Mode, RSB executes programs that you or another programmer have loaded into the Color Computer. RSB chooses what it will do next automatically, based on the program, instead of waiting for you to type in a command.

Once RSB enters Run Mode, it remains there until the computer encounters an instruction that tells it to return to Direct Mode.

If a program is running and you need to regain control of RSB right away, you can often do so by depressing the BREAK key.

Entering a Program

BASIC programs are divided into lines. Each line begins with a unique line number which must be an integer between 0 and 63999.

You enter a program by typing in all of the lines. No matter what order you type the lines in, RSB always arranges them from lowest to highest line number. For example, the typing sequence:

```
20 print "any lines to be"
10 print "RSB won't allow"
30 print "out of order."
```

results in the program:

```
10 PRINT "RSB won't allow"
20 PRINT "any lines to be"
```



```
30 PRINT "out of order."
```

Notice that RSB also converts lower-case command names to upper case.

Editing a Program

When you enter a line in Direct Mode, which begins with a line number, that line will modify the BASIC program in one of several ways.

If you have not already entered a line with this line number, the new line is inserted into the BASIC program just after the line with the next lower line number.

Entering just a line number, with no other data on the line, tells RSB to delete any line with that line number from the BASIC program.

If you enter a line that begins with the line number of an existing line, and there is data after the line number, RSB replaces the old line with the new line.

You can also use RSB's EDIT command to modify an existing line without retyping it. The EDIT command allows you to insert, delete, or replace text in any line of the BASIC program.

Program Execution

The RUN command causes RSB to leave Direct Mode and enter Run Mode. Once RSB enters Run Mode, it remains there until the program has finished executing.

For BASIC programs, execution always begins at the lowest numbered line and continues from line to line in sequential order unless a BASIC statement (e.g. GOTO) changes the order.

The RUN command allows you to execute either the BASIC program that is in memory, or a BASIC program stored in a disk file. For example, the command:

```
run
```

will execute the program that is in memory. To execute a program stored in a disk file, use:

```
run "program:3"
```

This example will load and run the program called program.bas that is stored in disk directory #3. Note that specifying a program name in the RUN command will erase any BASIC program already in memory.

RSB also provides a special form of the RUN command that allows you to call any OS9 command as a subroutine from within a BASIC program. For example, the program line:

```
1000 RUN "$free /d0"
```

will execute the OS9 FREE command from within a BASIC program.

Since RSB is also an OS9 command, you can even call one BASIC program from within another. A typical example would be:

```
1000 RUN "$rsb /d0/basic/menus.bas #8K"
```


Saving Programs

RSB will save programs in either ASCII or tokenized format. When loading a program, RSB automatically adapts to either format by examining the first two bytes of the file.

To save a program in tokenized format, use the normal form of the SAVE command:

```
save "program"
```

To save a program in ASCII format, add an 'A' after the SAVE command as follows:

```
save "program",A
```

Programs saved in ASCII format usually take up more disk space than tokenized progams. The advantage of ASCII format is that ASCII files are compatible with text editors and other software packages.

Leaving RSB

When you are done using RSB, you can return to the OS9 prompt by entering the DOS command.

NOTE: The DOS command does not save the BASIC program that is in memory. If you do not wish to lose the BASIC program that is in memory, be sure to SAVE it before using the DOS command.

Chapter 3

Data and Variables

Data and variables are the two classes of objects that RSB programs manipulate. Each class has two types: numeric, and string.

RSB can also manipulate groups of related variables, in the form of arrays.

Numeric Data

RSB allows you to represent numbers in BASIC programs as decimal, hexadecimal, or octal constants.

Decimal constants may be entered in integer, floating-point, or scientific notation. Here are some examples:

```
123456789       (integer)
3.14159         (floating-point)
-1.228E+6       (scientific)
```

Numbers larger than 999,999,999 or smaller than 0.01 are automatically displayed in scientific notation. The largest decimal number that RSB will accept is +1E+38, and the smallest positive (non-zero) number is +3E-39.

Hexadecimal constants are indicated by a leading &H symbol, and are always integer

values. A hexadecimal constant may have as many as 6 digits, so you can represent any number between 0 and 16,777,215. Typical hexadecimal constants are:

```
&HFF        (255)
&H4000      (16384)
&HFFFFFF    (16,777,215)
```

Octal constants are indicated by a & or &O symbol, and are also always integer values. An octal constant may have as many many as 8 digits, giving the same range of values as with hexadecimal constants. Here are some octal constants:

```
&377        (255)
&O12        (10)
&O77777777  (16,777,215)
```

Regardless of how you enter a numeric constant, it is always converted to binary scientific notation before RSB performs any operation on it.

String Data

A string is an ordered list of from 0 to 255 8-bit values. Strings may be used to represnt ASCII text, machine language programs, register values, or arbitrary binary data.

Strings are usually used to represent text. RSB allows you to define text strings easily, by enclosing the text in quotation marks. Here are some typical string constants:

```
"string constant"
```

```
"ABCEDFGHIJKLMNOPORSTUVWXYZ"
"JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC
```

Notice that the closing quotation mark has been left off of the third example. RSB lets you leave off the closing quote at the end of a line, as a short-cut.

Simple Variables

A simple variable is a storage place for a single text or numeric value. The value may be a constant, or may be the result of a calculation.

Every variable has a name and a type. The two-character name must begin with a letter (A-Z), but the second character may be either a letter or a digit (0-9). Certain names, like TO and AS, are not allowed because they conflict with a pre-defined RSB command word. Aside from these rules, you can choose any variable names that you like.

RSB will ignore any characters that come after the first two in a variable name. This allows you to make programs more understandable by using meaningful variable names. Consider which of these calculations is easier to understand:

```
T = .04 * B
TAX = .04 * BALANCE
```

The disadvantage of long variable names is that they make the program longer.

The type of a variable is assumed to be numeric, unless you specify a string

variable by placing a $ character at the end of the variable name. For example,

```
AX      (numeric variable)
AX$     (string variable)
```

Each numeric variable can store any legal numeric value, and you can change the value of a numeric variable at any time.

Each string variable can store any number of characters between 0 and 255. You can chage the length of a string variable at any time.

Array Variables

An array is an ordered list of objects, much like a string. The objects that make up an array may be either numbers or strings.

Every array has a name and a type, as with simple variables. Unlike simple variables, arrays also have dimensions and extents.

Arrays may have up to 255 dimensions, but the total number of array elements is limited by the amount of RAM allocated to RSB.

An array name is identified by a list of subscripts enclosed in parenthesis. Typical aray names are:

```
BOARD(1,5)      (numeric array)
DOC$(37)        (string array)
```


If RSB encounters an array that has not been defined with the DIM statement, it defaults the size of the array to one dimension with an extent of 11 elements.

Array subscripts begin at zero, and include the number specified as the extent of each dimension in the DIM statement. For example,

```
DIM A(22)
```

defines a 23 element array with elements numbered from 0 to 22.

Chapter 4

Expressions

You define calculations to RSB by including expressions in your programs. An expression is a combination of operators and operands.

Both data and variables (see Chapter 3) are operands. An entire expression, when enclosed in parenthesis, is also an operand to RSB.

The operators in an expression specify the type of calculation that RSB will perform on the operands. Typical operators specify addition, division, assignment, comparison, or other similar calculations.

Each operator has a precedence, which determines the order in which operations in an expression will be performed. A high-precedence operator may be evaluated prior to one with lower precedence, even though the low-precedence operator occurs first in an expression. This is explained in the Operator Precedence section of Chapter 4.

Arithmetic Operators

These operators perform mathematical calculations, and may only be used on numeric operands. There are two types of arithmetic operators: binary and unary.

Binary operators produce a result from two operands; unary operators produce a result from a single operand.

The binary arithmetic operators are:

=	Assignment
+	Addition
-	Subtraction
*	Multiplication
/	Division
	Exponentiation

Except for the assignment operator, all of the binary arithmetic operators are evaluated from left to right. For example,

```
C + B - A
```

is calculated as (C+B) - A, rather than as C + (B-A). But,

```
A = B = C
```

assigns the value of C to B, and then the new value of B to A. Note, however, that

```
A = (B=C)
```

will compare B to C, and set A according to the result. You cannot use the assignment operator inside of parenthesis.

The unary arithmetic operators are:

+	Assertion (no effect)
-	Negation (change sign)

These operators are evaluated from right to left. For example, the expression:


```
(-+B)
```

first computes the assertion of B and then computes the negation of B.

Relational Operators

The relational operators compare the value of two operands of the same type. Numeric operands are compared directly, and "dictionary order" is used to compare string operands.

All relational operators are binary. The relational operators are:

=	Equal To
<>	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

The relational operators are evaluated from left to right.

The result of a relational operation is always numeric. A true relation has the value (-1), and a false relation has the value (0).

Strings are compared as if you were to them up in an English dictionary. The closer a word or phrase is to the end of the dictionary, the greater its value. For example, the relation:

```
"dog" > "cat"
```

is true because "dog" would occur after "cat" in a dictionary of English.

Logical Operators

Logical operators calculate bit-by-bit logical operations on their operands.

The binary logical operators are:

AND	Bit-wise AND
OR	Bit-wise OR

These operators are evaluated from left to right.

The unary logical operators are:

NOT	Bit-wise NOT (inversion)

These operators are evaluated from right to left.

Only numeric operands, with values in the range -32768 to +32767, may be used with logical operators.

String Operators

String operators perform operations on string operands, producing a string result.

These are the string operators:

=	Assignment
+	Concatenation (join strings)

The string assignment operator is evaluated from right to left, and follows the same

rules as numeric assignment. The string concatenation operator produces a string consisting of the left-hand operand followed immediately by the right-hand operand; it is evaluated from left to right.

Operator Precedence

The Table summarizes the precedence of each operator. Operators at the top of the Table have higher precedence than those at the bottom. Operators on the same line have equal precedence.

(...)	Any	Parenthesis
NOT, +, -	Numeric	Unary operators
+	String	Concatenation
<,>,= etc	String	Any relational
	Numeric	Exponentiation
*, /	Numeric	Mult., Divide
+, -	Numeric	Add, subtract
<,>,= etc	Numeric	Any relational
AND, OR	Numeric	Binary logical
=	Any	Assignment

Here's how you use the precedence table. For the expression:

```
3 AND 4 > 3
```


we see that the precedence of the numeric '>' operator is higher than that of the 'AND' operator. This means that RSB will evaluate the expression as if it were written:

```
3 AND (4>3)
```

Functions

RSB includes many built-in functions, and you can use the DEF statement to create custom functions when your programs need them.

Functions accept one or more arguments, and produce a single result. The arguments need not all be of the same type. Some functions return numeric values, and others return string values.

The result of a function is an operand that can be used as part of an expression. Typical RSB built-in functions are:

```
SIN(x)          (trigonometric sine)
ABS(x)          (absolute value)
LEFT$(a$,p)     (left substring)
```

References to functions look very much like references to arrays. You can always identify a function by looking up its name in the command reference in Appendix A.

Chapter 5

Files

RSB provides powerful file I/O commands that allow you to access files stored on any OS9 device. Like Disk Extended Color BASIC, RSB supports both sequential and random files. RSB also allows you to treat any OS9 device as a file.

Drive Numbers

RSB takes advantage of OS9's Unified I/O System to give you access to any of your system's storage devices from BASIC. RSB is compatible with all types of OS9 RAM disks, floppy drives, and hard drives.

DECB allowed you to select one of four disk drives by using a number between 0 and 3. Typical DECB disk commands looked like this:

```
DIR 1                  (accessed disk #1)
OPEN "I",1,"TMP:3"     (accessed disk #3)
```

The number 1 in the first command, and the number 3 in the second command, are both "drive numbers" that specify which floppy drive to use.

With RSB, drive numbers take on a new meaning: rather than selecting a floppy disk drive, they select an OS9 directory.

You can have hundreds of directories, located on many different devices, under OS9. RSB provides a new command, OPEN DRIVE, that lets BASIC programs access any directory as any drive number. Here's how it works:

```
100 OPEN DRIVE 3,"/d0/basic/games"
```

This command tells RSB that any future references to drive 3 should access the directory /d0/basic/games. OPEN DRIVE can set the directory for any drive number 0-3. A BASIC program can even set several drive numbers to use the same directory.

Sequential Files

OS9 does not divide files into various types, but RSB will treat any file opened with the "I" or "O" option as a sequential file.

A sequential file is the simplest type of file supported by RSB. This type of file is just a long stream of data.

To read anything from a sequential file, a program must also read everything that comes before it in the file. When a program writes to a sequential file, all data after the written item is erased.

You can't update a sequential file. Instead, RSB programs create a new file and copy the contents of the old file to it. Any updates must be made as the file is being copied.

Sequential files can be of any length.

Random Files

RSB will treat files opened with the "R" or "D" options as random files.

Random files are organized as a list of fixed-size records. Programs can read or write any random file record, at any time, without having to read or change the rest of the file.

You can think of a random file record as an electronic index card. Index cards come in various sizes, each size holding a certain amount of information. You don't have to use all of the available space on an index card, but if you aren't using all of it you may need a different size of card.

The OPEN statement specifies how many bytes of information can be stored in each record of a random file. Each open random file has its own record size; for example, one random file may be made up of 37 byte records, while another may use 3000 bytes per record.

Each random file may have as many as 65,000 records.

Devices as Files

Without OS9, DECB had very limited support for serial devices. RSB takes advantage of OS9's Unified I/O System to provide enhanced support for this type of

device.

RSB allows you to open serial, or SCF, devices as if they were sequential files. RSB's sequential file I/O commands can also read, write, and check the status of SCF devices.

For example, the line:

```
300 OPEN "O",1,"/P"
```

will open the device "/P" for sequential output. If "/P" is a printer, the line:

```
310 PRINT #1,"Hello"
```

will send the word "Hello" to the printer.

Chapter 6

Text, Graphics, and Sound

Most BASIC programs generate text or graphics, or both. Since RSB runs in an OS9 window, text and graphics are displayed in the current window.

Level 2 OS9 provides two types of windows: VDG windows, which are compatible with the CoCo 2, and true windows, which allow high-resolution text and graphics. RSB will run in either type of window.

Some RSB commands operate differently in VDG windows than in true windows. For example, high-resolution text and graphics commands are not allowed in VDG windows.

Music is also an important part of many BASIC programs. RSB's PLAY and SOUND work like their DECB counterparts, but can also be used to control a Speech/Sound PAK or other sound generation hardware.

Text Attributes

The CoCo 3 allows you to use 32, 40, or 80 column screens from BASIC. Under RSB, you must be running in a true window to use 40 or 80 column screens. VDG windows are limited to 32 columns of text.

On 40 and 80 column screens, BASIC programs can use the ATTR command to control forground color, background color, blinking and underlining for each screen location. This command can also be used with 32 column screens if RSB is running in a true window, but will generate ?HR.ERROR if used on a VDG screen.

When OS9 creates a VDG screen, it will enable either true lower case or inverse video based on the TYPE byte of the window descriptor. Both types of VDG windows are compatible with RSB.

Under DECB, the 32 column text screen was always located at address $400. OS9 alloates RSB's text screen from system memory, so the address of the screen is not available to RSB programs. RSB uses the area from $400-$5FF for other purposes. Programs that access the 32 column text screen directly must be modified before they can be used with RSB.

RSB's HSTAT command can be used to determine the current cursor position on a 40 or 80 column screen. Since OS9 does not provide a facility to read the character under the cursor, HSTAT does not return the character code or character attributes.

Block Graphics

Some BASIC programs use block graphics to create simple, colorful shapes mixed with 32 column text.

On a 32 column VDG screen, character codes greater than 127 produce standard

CoCo 2 block graphics. RSB does not support block graphics on true windows.

Low-Res Graphics

Low-resolution graphics provide up to 256 x 192 graphic points in two colors, or up to 128 x 192 graphic points in four colors.

For maximum compatibility, you should use a VDG screen to run programs that use low-res graphics.

RSB will automatically convert low-res graphics commands into high-res commands when using a true window. This sometimes produces surprising results. One common example is that a translated DRAW statement may leave a 1 pixel gap in the resulting design -- wreaking havoc with subsequent PAINT statements. A simple edit of the DRAW statement usually fixes the problem.

OS9 always allocates a full 6K buffer for VDG (low-res) graphics. The equivalent DECB graphics modes are PMODE 3 and 4. RSB emulates PMODE 0, 1, and 2 by converting commands as follows:

Actual Command	Translation
PMODE 0	PMODE 4
PMODE 1	PMODE 3
PMODE 2	PMODE 4
PCOPY	equivalent PCOPY
PCLEAR	None - edit program

Programs that use the PCLEAR command must

be modified to reserve memory as if they were using PMODE 4.

Programs may allocate zero, one, or two graphics screens. Unlike DECB, RSB does not allocate graphics screens from BASIC program memory; however, there must be enough room in RSB's 64K memory map to allocate the number of screens requested.

The memory for low-res graphics screens is not included in the memory size that you specify on the RSB command line. Entering:

```
rsb #24K
```

gives RSB 24K of storage. RSB itself takes up 32K, so 8K remains for low-res graphics screens.

RSB automatically converts the starting page number for PMODE graphics displays to the nearest lower or same page that is on an even full-screen boundary. For example, PMODE 3,2 is identical to PMODE 3,1 because graphics pages 1 and 3 are both part of the first PMODE 3 graphics screen.

When running in a VDG window, RSB uses DECB's original routines to perform all low-res graphics commands. Apart from the translations mentioned above, all low-res graphics commands perform identically under RSB and DECB when a VDG window is used.

When running in a true window, RSB translates all low-res graphics commands to similar commands that are compatible with a 16 color, 320 x 192 graphics window. Only the left-most 256 pixels of this window are used; the right-hand edge of the window is

cleared to the background color.

Colors are converted to the matching high-res color set. Coordinates are scaled to OS9's 640 x 192 standard range.

Some commands cannot be translated from low-res to high-res, including:

GET PCOPY PUT

Executing any of these commands in a true window will produce an ?HR ERROR.

When using PMODE-type graphics commands on a true window, RSB will display graphics as soon as any graphics command draws on the graphics screen, or whenever the SCREEN 1 command is executed.

High-Res Graphics

One of the best features of the CoCo 3 is its high-resolution graphics capability.

RSB uses OS9's powerful windowing system to draw high-resolution graphics. All graphics commands are implemented with standard OS9 graphics commands and service calls. A key benefit of this technique is that RSB high-res graphics will work on any future graphics devices that meet the OS9 standard, or on an external graphics terminal.

There are a few important differences between OS9's and DECB's high-res graphics commands.

First, all OS9 hi-res graphics commands use a 640x192 coordinate system, even on screens that are only 320 pixels across. DECB uses absolute coordinates on all high-res screens. RSB handles this difference by scaling coordinates on 320x192 screens. DRAW commands that produce closed figures under DECB may leave a 1 pixel gap under RSB, due to scaling of the endpoints. For example, the command:

```
HDRAW "U10;L10;D10;R10"
```

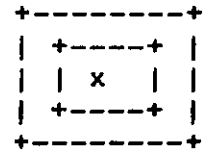
must be changed to:

```
HDRAW "U10;L10;D10;R11"
```

if you intend to use it on a 320 pixel RSB high-res graphics screen.

Second, OS9's hi-res flood-fill command works differently than DECB's HPAINT. The DECB HPAINT command fills with a specified color until a specified border color is encountered; RSB's HPAINT changes the color of all adjacent pixels, that are the same color as the starting point, to a specified color. If you specify a border color with RSB's HPAINT, it is ignored.

In many cases the two HPAINT commands produce identical results, but here's an example that illustrates the difference:



Suppose that the outer box is white, the

inner box is blue, and the area inside each box is red. A DECB HPAINT command that specifies a white border will fill the entire outer box and erase the inner box, but the same command in RSB will just fill the inner box.

No-Halt Sound

RSB uses standard OS9 SS.TONE system calls to generate sound for PLAY and SOUND commands. These commands are normally sent to standard output, but the RSB environment file allows you to specify any sound producing device.

If no sound device is specified, RSB uses the Windowing System's built-in tone generation software. This software slows down multitasking significantly during tone generation.

You can use the Speech / Sound PAK, Super Voice, or other no-halt sound-producing hardware with RSB. To do so, you must install an OS9 driver for the sound device and list it in the environment file. Note that your sound device driver MUST support the SS.Tone service call.

Appendix A

Commands Reference

RSB programs are made up of commands that direct the actions of your computer. Some commands allow parameters or options that select one of several actions.

Options and parameters are usually expressions (see Chapter 4). An expression performs a calculation, and may call one or more RSB functions in the process.

Both commands and functions are written as short groups of letters, or keywords. Disk Extended Color BASIC keywords are described in your Color Computer 3 Extended BASIC and Color Computer Disk System manuals. This Chapter describes several commands that are new in RSB, and commands that work differently under RSB than under DECB.

Reference Format

The listing for each command is divided into several sections:

Command name
Syntax (spelling, option arrangement)
Function (what the command does)
Parameters
Return Value
Notes (special features / requirements)

Examples

Commands and functions are intetermixed. All keywords are arranged in alphabetical order. Some keyword listings vary slightly from the standard reference format; for example, Return Value is listed only for functions.

Command Name

This is the name of the command or function, and a brief description of it. Commands and functions are arranged in alphabetical order.

Syntax

Writing an RSB command is like writing a sentence in English; you have to arrange the keywords in a certain order, use the correct punctuation, and so on, if you want the computer to understand your command.

The syntax of a command is a set of instructions for writing that command correctly.

Upper-case letters indicate a word that you must enter exactly as it is written. Lower-case letters mark places where you can fill in an option or parameter value; the lower case letters only describe the type of information that you must supply.

Several special symbols are used in command syntax, to make the instructions shorter. These symbols are:

[ ] Anything betweeen these brackets is optional. You don't type the brackets as part of the command, just what's between them. Brackets can be nested to indicate options of options.

... This symbol indicates that you can repeat the previous item as many times as you like.

| When you see items separated by this symbol, you must choose only one of the items. The list of choices is enclosed in {} brackets.

Function

A general description of the command's operation is presented here. You should also refer to the Notes section for information about any special features or requirements of the command.

Parameters

Some commands (like INKEY$) are made up of a single keyword. More commonly, you must provide additional information with a command in order to completely specify its action. This additional information can have several parts, and each part is called a parameter.

The parameters section gives you full descriptions of the same command parameters that were described as single words in the command syntax. The description includes

both ranges of legal values and default values.

Return Value

All RSB functions produce a result, or return value. This value may be a string or a number; it may have a particular range of legal legal values or length.

The Return Value section tells you what type of information a function returns. Commands do not produce a return value, so this section is omitted for commands.

Notes

Advanced programmers sometimes need to know more about a command than is provided in the Function section. Very detailed or noteworthy information about a command is listed in the Notes section.

For example, if a command disables OS9 interrupts, this information is listed in the Notes section rather than the Function section.

Examples

This section provides at least one example of an actual command or function, with all parameters filled in. It also describes the action performed by that specific command.

AUDIO Control Cassette Port Audio

Syntax:

AUDIO {ON|OFF}

Function:

This command is provided for compatibility with DECB. It does not perform any function under RSB.

Examples:

AUDIO ON

This command is ignored.

BACKUP Duplicate a Disk

Syntax:

BACKUP s_drive TO d_drive

Function:

This command copies an entire disk to another disk of the same size.

Parameters:

s_drive Source drive number, 0-3

d_drive Destination drive number, 0-3

Notes:

RSB calls OS9's BACKUP utility to do the actual work.

The source and destination drive names are obtained from the directory names for drives 0-3. You must use full path names for the directories or BACKUP will not be able to determine the drive name.

If the source and destination drive numbers are identical, RSB initiates a single-drive backup.

Examples:

BACKUP 0 TO 1

Copies everything on drive 0 to drive 1.

CLEAR Erase Variables & Configure

Syntax:

```
CLEAR [s_mem[,m_limit]]
```

Function:

Erases all variables and arrays. Can also be used to allocate memory for string variables and machine language subroutines by specifying optional parameters.

Parameters:

s_mem	Amount of memory to allocate for string variables. Default value is no change.
m_limit	Highest RAM address that RSB may use for program and data storage. This address must be within the RAM allocated to RSB. Default value is no change.

Notes:

At reset, RSB automatically allocates 200 bytes of string space. Programs that manipulate strings usually need more than 200 bytes. A good rule for guessing how much string space you need is to add up the maximum length of each string, plus 3 bytes per string, plus 10%.

You can safely load machine language subroutines or data at the first address after m_limit, as long as whatever you load ends before the beginning of RSB. You can find out the beginning address of

RSB with the line:

```
RS = 256*PEEK(&H5E0)+PEEK(&H5E1)
```

Examples:

```
CLEAR 1024
```

Erases all variables, and reserves 1024 bytes for strings.

CLOAD Load Program from Cassette

Syntax:

```
CLOAD [file_name]
```

Function:

Loads a BASIC program from OS9's cassette device.

Parameters:

file_name	A string that names the file to be loaded. If no name is given, RSB will try to load the first BASIC program file stored on the cassette device.

Notes:

RSB reads the name of the cassette device from the environment file at startup.

CLOAD and other RSB cassette I/O commands access the cassette device via OS9 system calls. Any device with a suitable device driver can be used as the cassette device.

The RSB package does not include a cassette device driver. RSB's cassette I/O driver requirements are listed in Appendix E.

Example:

```
CLOAD "BRIDGE"
```

Loads the BASIC program named BRIDGE

from the cassette device.

CLOADM Binary Cassette Load

Syntax:

CLOADM [file_name][,offset]

Function:

Loads a machine language program or data file from OS9's cassette device.

Parameters:

file_name	A string that names the file to be loaded. If no name is given, RSB will try to load the first binary file stored on the cassette device.
offset	This value is added to each load address in the binary file, effectively loading the file higher or lower in memory than its default address. The legal range is 0-65535. The default value of offset is 0.

Notes:

If the sum of the offset and a load address is greater than 65536, RSB subtracts 65536 to obtain a legal address.

You are responsible for making sure that CLOADM does not destroy any information needed by RSB or OS9. Be sure to use the CLEAR command to reserve memory for machine language or binary data files before using CLOADM.

See also CLOAD.

Example:

```
CLOADM "SCREEN",16384
```

Loads the binary file named SCREEN from the cassette device, adding 16384 to every load address in the file.

COPY Copy a File

Syntax:

COPY s_name [TO d_name]

Function:

This command creates a copy of a single file.

Parameters:

s_name	Source file name
d_name	Destination file name. Defaults to same as s_name.

Notes:

RSB calls OS9's COPY utility to do the actual work.

If the destination file name is omitted, RSB initiates a single-drive copy command.

Examples:

```
COPY "TEST.BAS" TO "TEST.OLD:1"
```

Copies the file TEST.BAS from the current defult drive to drive 1, giving the copy the name "TEST.OLD".

CSAVE Save Program to Cassette

Syntax:

CSAVE file_name[,A]

Function:

Saves a BASIC program to the cassette device.

Parameters:

file_name	The name to give the file when saving it.
A	If specified, save the file in ASCII format. Otherwise the file is saved in tokenized format.

Notes:

See also CLOAD.

Example:

```
CSAVE "PROG1",A
```

Saves whatever BASIC program is in memory to the cassette device in ASCII format, with the name PROG1.

CSAVEM Binary Cassette Save

Syntax:

```
CSAVEM file_name,start,end,exec
```

Function:

Saves a machine language program or data file to OS9's cassette device.

Parameters:

file_name	The name to give the file when saving it.
start	The lowest address to be saved to the file.
end	The highest address to be saved to the file.
exec	The address at which to begin executing the file when it is reloaded.

Notes:

The exec parameter must be present in every CSAVEM command. When you reload the file, the default address of the EXEC command is automatically set to whatever address you used in the CSAVEM command. Even if you are just saving binary data to the cassette, you must still specify a value for the exec parameter.

See also CLOAD.

Example:


```
CSAVEM "DATA1",&H3000,&H3FFF,&H0000
```

Saves the binary data between addresses &HC000 and &HC00F (inclusive) to cassette data file named DATA1.

DIR Show Directory

Syntax:

DIR [drive]

Function:

This command displays the directory of the specified drive.

Parameters:

drive	Drive number, 0-3. Defaults to the default drive if not specified.

Notes:

RSB calls OS9's DIR utility to do the actual work.

The source and destination drive names are obtained from the directory names for drives 0-3.

Example:

DIR 2

Display the directory for drive 2.

DOS Exit to OS9

Syntax:

DOS

Function:

Closes all files, erases any BASIC or machine language programs from memory, stops RSB, and returns control to OS9.

Parameters:

Notes:

Early versions of Disk BASIC did not provide a DOS command. The RSB installer automatically adds the DOS command to your version of Disk BASIC if necessary.

Example:

```
9999 DOS
```

Returns control to OS9 when the BASIC program executes line 9999.

DRAW Draw a Low-Res Figure

Syntax:

```
DRAW f_info
```

Function:

Draws a figure on the low-res graphics screen. The figure is made up of line segments described by the string f_info.

Parameters:

f_info	A DRAW parameter string as described in Color Computer 3 Extended BASIC.

Notes:

All coordinates used in DRAW commands are scaled when using DRAW on a true window.

Coordinate scaling can cause 1 pixel "holes" in the border of figures that do not have overlapping endpoints, leading to problems with subsequent PAINT commands. You can correct this by editing the figure so that the beginning and end points of the line overlap by at least 1 unit.

On a true window, executing DRAW (or any other graphics command) while a text screen is displayed will erase the text screen and display the graphics screen.

Example:

```
DRAW "U10;D10;L10;R11"
```


Draws a box, 11 units on a side, on the low-res graphics screen. Note that the length of the final segment is 11, rather than 10, to make the endpoints of the figure overlap.

DSKI$ Raw Disk Read

Syntax:

DSKI$ drive,track,sector,bfr1,bfr2

Function:

Reads an absolute sector from a disk.

Parameters:

drive	Drive number, 0-3.
track	Track number, 0-191
sector	Sector number, 0-63
bfr1	Name of string variable to store first 128 bytes of sector data.
bfr2	Name of string variable to store second 128 bytes of sector data.

Notes:

This command is not supported with the standard CC3Disk OS9 floppy driver, but is supported by several third party floppy drivers.

Direct disk reads use the SS.DREAD status call. Check your third-party disk driver manual for further information about SS.DREAD.

For floppy drives, the number of sides is determined by examining the PD.OPT section of the drive path descriptor, and

not by reading LSN0. DSKI$ will not work correctly if you try to use a single-sided disk in a double-sided drive.

DSKI$ is not supported on hard drives.

Example:

```
DSKI$ 0,34,0,A$,B$
```

Read the first sector of track 34 to the variables A$ and B$.

DSKINI Format Disk

Syntax:

DSKINI drive

Function:

This command erases all information from a disk and prepares it for use under OS9.

Parameters:

drive Drive number, 0-3.

Notes:

RSB calls OS9's FORMAT utility to do the actual work.

The drive name is obtained from the directory names for drives 0-3.

Example:

DSKINI 2

Format disk where directory #2 is located.

DSKO$ Raw Disk Write

Syntax:

DSKO$ drive,track,sector,bfr1,bfr2

Function:

Writes an absolute sector to a disk.

Parameters:

See DSKI$

This command is not supported with the standard CC3Disk OS9 floppy driver, but is supported by several third party floppy drivers.

Direct disk writes use the SS.DWRIT status call. Check your third-party disk driver manual for further information about SS.DWRIT.

DSKO$ is not supported on hard drives.

Example:

DSKO$ 0,34,0,A$,B$

Write the first sector of track 34, using the data in variables A$ and B$.

EOF Check for End of File

Syntax:

```
EOF(device)
```

Function:

Return a value indicating whether or not the end of a file has been reached.

Parameters:

device	The number of the device to be checked. Must be -2 - 15.

Return Value:

-1 if at end of file, 0 if not

Notes:

On SCF-type devices, EOF first checks the device's one byte cache and returns 0 if there is data in the cache. If there is no data in the cache, EOF uses the SS.EOF status call to check for end of file.

Example:

```
110 IF EOF(1) THEN 130
120 READ #1,A$
```

Line 110 checks for end of file on device 1, and skips line 120 if end of file is detected.

EXEC Call Machine Language Routine

Syntax:

EXEC [address]

Function:

Calls a machine language subroutine.

Parameters:

address — The starting address of the machine language subroutine. If no address is specified, the address set by the most recent LOADM or CLOADM command is used.

Notes:

No particular register values are passed to the machine language subroutine.

All OS9 programs, including RSB, are position independent and may be loaded at any memory address. BASIC programs must take an extra step when using the EXEC command to call a routine that is part of RSB.

For example, the DECB routine that reads the joysticks is at address &HA9DE. A DECB program could call this routine by executing the command:

EXEC &HA9DE

Under RSB, you must modify the command to account for OS9's position-independent loading. The RSB version of this command

is:

```
A=256*PEEK(&H5E0)+PEEK(&H5E1):
B=&H7FFB:
EXEC A+(&HA9DE-B)
```

'A' is the starting address of RSB. 'B' is a "magic number" that, when subtracted from a DECB address, gives the relative location of the same address in RSB.

You can also use EXEC to call machine language programs that were loaded with the LOADM, CLOADM, or POKE commands. Be sure to use the CLEAR command to reserve memory for machine language routines before loading them into memory.

Machine language routines do not have to be position independent, but they must use OS9 system calls for all I/O.

There are significant internal differences between DECB and RSB. Machine language routines that attempt to modify BASIC (e.g. add new commands or change command operation) will probably not work with RSB, and may crash your computer.

Example:

```
100 CLEAR 255,&H3FFF
110 POKE &H4000,&H39:'RTS instruction
120 EXEC &H4000
130 PRINT "ALL DONE!"
```

Line 100 reserves RSB's memory beyond address &H3FFF for machine language routines. This line also reserves an arbitrary amount of RAM, 255 bytes, for

strings.

Line 110 uses the POKE command to store a very simple machine language routine at address &H4000. Note that this is the first available address defined by the CLEAR command in line 100.

Line 120 uses the EXEC command to call the routine that was set up in line 110. When the routine exits, it returns control to RSB.

Line 130 has been added to show that the BASIC program continues to run after completely executing the machine language routine.

FREE Calculate Free Disk Space

Syntax:

```
FREE(drive)
```

Function:

This function is provided for compatibility with DECB. It does not perform any real function under RSB.

Parameters:

drive	Drive number to check (0-3)

Return Value:

Always returns 1.

Examples:

```
A = FREE(0)
```

Returns a value of 1.

GET Copy Low-Res Graphic Rectangle

Syntax:

GET (sx,sy)-(ex,ey),array[,G]

Function:

Copies a rectangular area of the low-res graphics screen to an array. You can then use the PUT command to duplicate the rectangle anywhere on the graphics screen.

Parameters:

sx	X-coordinate of one corner of the rectangle, 0-255
sy	Y-coordinate of one corner of the rectangle, 0-191
ex	X-coordinate of the opposite diagonal corner of the rectangle, 0-255
ey	Y-coordinate of the opposite diagonal corner of the rectangle, 0-191
array	The name of a two-dimensional array that is big enough to hold the graphic rectangle
G	Enables full graphic detail. This option is required when using PMODE 3 or PMODE 4.

Notes:

The GET command generates a ?HR ERROR when used on a true window.

Each screen pixel is stored in a single array element. You need a 12 x 3 array to store a 12 x 3 rectangle.

When using GET and PUT together, you must use the same PMODE for both the GET and the PUT.

Example:

```
100 DIM A(19,2)
110 GET (0,0)-(19,2),A
120 PUT (100,100)-(119,102),A
```

Line 100 declares an array big enough to hold a 20 x 3 graphic rectangle. Note that the DIM command reserves elements beginning at subscript 0, so line 100 actually declares a 20 x 3 array.

Line 110 copies the 20 x 3 graphic array described by the diagonal line from (0,0)-(19,2) into array A.

Line 120 duplicates the original rectangle at another screen location.

HBUFF Declare Hi-Res Graphic Buffer

Syntax:

```
HBUFF buffer,size
HBUFF 0
```

Function:

Reserves memory for use by subsequent HGET and HPUT commands.

Parameters:

buffer	A buffer number in the range 1-253
size	The number of bytes to reserve for the buffer, minus one, in the range 0-32767

Notes:

HBUFF 0 erases all buffers and returns their memory to OS9.

Hi-res graphics buffers, unlike the arrays used in low-res graphics, store an exact image of the binary screen data. The number of bytes needed to store a hi-res graphic rectangle depends on which HSCREEN you are using. You can find a detailed explanation of how to calculate the size of a hi-res buffer in Color Computer 3 Extended BASIC.

Once you have allocated a buffer, the only way to change its size is to use the HBUFF 0 command. This command erases and deallocates all hi-res graphic buffers that your program has allocated.

The first time you use a buffer number in an HGET command, OS9 will automatically allocate the correct amount of memory to that buffer. DECB does not allocate buffers automatically, so programs that take advantage of RSB's automatic buffer allocation will not run correctly under DECB.

Example:

```
HBUFF 3,65
```

Reserves 65 bytes for hi-res buffer 3.

HDRAW Draw a Hi-Res Figure

Syntax:

```
HDRAW f_info
```

Function:

Draws a figure on the hi-res graphics screen. The figure is made up of line segments described by the string f_info.

Parameters:

f_info	An HDRAW parameter string as described in Color Computer 3 Extended BASIC.

Notes:

All coordinates used in HGET commands are scaled in HSCREEN 1-2.

Coordinate scaling can cause 1 pixel gaps in the border of figures that do not have overlapping endpoints. If you plan to use HPAINT to fill an HDRAW figure, make sure the HDRAW endpoints overlap.

Example:

```
HDRAW "U10;D10;L10;R11"
```

Draws a box, 11 units on a side, on the hi-res graphics screen. Note that the length of the final segment is 11, rather than 10, to make the endpoints of the figure overlap.

HGET Copy Hi-Res Graphic Rectangle

Syntax:

HGET (sx,sy)-(ex,ey),buffer

Function:

Copies a rectangular area of the hi-res graphics screen to a hi-res graphics buffer. You can then use the PUT command to duplicate the rectangle anywhere on the graphics screen.

Parameters:

sx	X-coordinate of one corner of the rectangle, 0-639 or 0-319
sy	Y-coordinate of one corner of the rectangle, 0-191
ex	X-coordinate of the opposite diagonal corner of the rectangle, 0-639 or 0-319
ey	Y-coordinate of the opposite diagonal corner of the rectangle, 0-191
buffer	The number of the hi-res buffer to be used to store the graphic rectangle.

Notes:

All coordinates used in HGET commands are scaled in HSCREEN 1-2.

HGET stores graphic data in the buffer in binary format, exactly as it is stored

in the displayed screen image.

When using HGET and HPUT together, you must use the same HSCREEN for both operations.

Example:

```
HGET (100,100)-(119,119),3
```

Copies the hi-res rectangle described by the diagonal line from (100,100)-(119, 119) to hi-res buffer 3.

HPAINT Hi-Res Graphics Flood Fill

Syntax:

HPAINT (x,y)[,p_color,b_color]

Function:

Changes the color of an area on the hi-res graphics screen.

Parameters:

x	Starting X-coordinate, 0-639 or 0-319
y	Starting Y-coordinate, 0-191
p_color	Paint color, 0-15. Defaults to current background color
b_color	Dummy argument provided for compatibility with DECB.

Notes:

HPAINT determines the current color of the point at (x,y) on the hi-res graphics screen. It then changes the color of that point and all adjacent points of the same color to a new color, p_color.

RSB's HPAINT operates differently than DECB's HPAINT. The RSB and DECB commands produce equivalent results in most cases.

Coordinates used in HPAINT commands are scaled in HSCREEN 1-2.

Colors used in HPAINT commands are automatically adjusted to the legal range

of colors for the current HSCREEN.

Example:

```
HPAINT (23,57),3,0
```

Changes the color of the area that includes point (23,57) to color 3.

HPRINT Hi-Res Text Output

Syntax:

```
HPRINT (x,y),text
```

Function:

Displays text on the hi-res graphics screen.

Parameters:

x	Starting column number, 0-79 or 0-39
y	Row number, 0-23
text	String value to display

Notes:

HPRINT will only display a single line of text. Any text that extends past the right-hand edge of the graphics screen is discarded.

Text is displayed in the currently selected OS9 graphic text font, using the current graphic foreground and background colors.

There are 80 text columns in HSCREEN 3-4, and 40 text columns in HSCREEN 1-2. All HSCREENs have 24 text lines.

Example:

```
100 HSCREEN 1
110 HPRINT (17,12),"HELLO!"
```


Line 100 selects a hi-res graphics mode that provides 40 x 24 text.

Line 110 prints the word, HELLO!, in the center of the hi-res graphics screen.

HSTAT Get Text Cursor Information

Syntax:

```
HSTAT c_var,a_var,x_var,y_var
```

Function:

Return the coordinates of the cursor on 40 and 80 column text screens.

Parameters:

c_var	Dummy string variable name for compatibility with DECB
a_var	Dummy numeric variable name for compatibility with DECB
x_var	Name of numeric variable to be set to the current cursor column number
y_var	Name of numeric variable to be set to the current cursor row number

Return Value:

c_var	Set to a single NULL character
a_var	Set to zero
x_var	Set to the current cursor column number, 0-39 or 0-79
y_var	Set to the current cursor row number, 0-23

Notes:

Unlike it's DECB counterpart, RSB's HSTAT command does not return the character code or attributes for the current cursor position.

Example:

```
100 LOCATE 10,0
110 HSTAT C$,A,X,Y
120 PRINT X,Y
```

Line 100 moves the text cursor to column 10 on text line 0.

Line 110 sets the variables X and Y to the current coordinates of the text cursor. The result is X=10, Y=0 due to line 100.

Line 120 displays the result of the HSTAT command.

JOYSTK Read Joystick or Mouse

Syntax:

```
JOYSTK(j_number)
```

Function:

Return the current horizontal or vertical position of one of the joysticks.

Parameters:

j_number — Indicates which axis and which joystick are to be read.

0	Horizontal, right joystick
1	Vertical, right joystick
2	Horizontal, left joystick
3	Vertical, left joystick

Return Value:

Number, 0-63

Notes:

The joystick values are only updated when both of two conditions are true:

1) RSB's window is the active display window.

2) The BASIC program calls JOYSTK(0)

Note that JOYSTK(0) reads the joystick values for both axes of each joystick. Subsequent calls to JOYSTK(1-3) return whatever values were read at the last call to JOYSTK(0).

The right joystick may be either an actual joysick or a mouse. The left joystick must be an actual joystick. The environment file tells RSB which type of device is connected to the right joystick port.

When using a mouse, RSB scales the horizontal and vertical coordinates to the normal joystick range of 0-63.

You can also use the entire range of the mouse by examining RSB's mouse status buffer ($600-$61F) after calling JOYSTK. This buffer contains the SS.Mouse packet, as defined in the Level 2 OS9 Technical Reference

Example:

```
100 'Read Hi-Res Mouse to X,Y
110 P=&H600:'Mouse buffer address
120 A=JOYSTK(0):'Read mouse
130 IF PEEK(P)=0 THEN RETURN
140 X=256*PEEK(P+24)+PEEK(P+25)
150 Y=PEEK(P+27):'Set new location
160 RETURN
```

This short subroutine reads the hi-res mouse coordinates to variables X and Y. Line 130 was added so that X and Y are not updated unless the window is active.

KILL Delete a File

Syntax:

KILL file

Function:

This command deletes the specified file.

Parameters:

file Name of the file to delete

Notes:

RSB calls OS9's DEL utility to do the actual work.

Example:

```
KILL "PROG.BAS:3"
```

Delete the program PROG.BAS from directory #3.

LOAD Load Program

Syntax:

LOAD file_name[,R]

Function:

Loads a BASIC program into memory.

Parameters:

file_name	A string indicating the name of the program file to be loaded.
R	If specified, RSB will RUN the program automatically after loading it.

Notes:

RSB automatically determines whether the file is in tokenized or ASCII fomat.

You can load a program from any device by using a full OS9 pathname as file_name.

The file name extension defaults to .BAS

Example:

```
LOAD "BRIDGE:3"
```

Loads the program named BRIDGE.BAS from directory #3.

LOCATE Move Text Cursor

Syntax:

```
LOCATE x,y
```

Function:

Move both the text cursor and the print position to the specified text row and column.

Parameters:

x	Desired column number, 0-31, 0-39, or 0-79
y	Desired row number, 0-15 or 0-23

Notes:

The LOCATE command works on all text screens, including 32 column VDG screens.

Example:

```
100 LOCATE 0,0
110 PRINT "NO PLACE LIKE HOME"
```

Line 100 moves the text cursor to column 0 on text line 0.

Line 110 prints a text message at the current cursor location.

LOF Check for Data Available

Syntax:

```
LOF(device)
```

Function:

Return a value indicating whether or not data is available from a device.

Parameters:

device	The number of the device to be checked. Must be -2 - 15.

Return Value:

See below.

Notes:

On SCF-type devices, LOF returns -1 if data is available and 0 if data is not available. LOF first checks the device's one byte cache and returns -1 if there is data in the cache. If there is no data in the cache, LOF uses the SS.READY status call to check for data.

LOF returns the number of records in a random file, or the number of bytes in a sequential file, when used on RBF devices.

Example:

```
110 IF A>LOF(1) THEN 130
120 GET #1,A
```

Line 110 compares the variable, A,

to the number of records in random file #1. If A is greater than the number of records, line 120 is skipped.

MOTOR Control Cassette Motor

Syntax:

```
MOTOR {ON|OFF}
```

Function:

Issues a motor control command to the cassette device.

Parameters:

ON	Turn the motor on
OFF	Turn the motor off

Notes:

The RSB software package does not include a cassette device driver, but RSB can access compatible cassette drivers.

The MOTOR command issues an I$SetStt call to the cassette device, with the following register values:

```
Reg-B = SS.ComSt
Reg-Y = $0008 (ON) or $0000 (OFF)
```

The path number is obtained at startup, by opening the cassette path specified in the environment file.

Example:

```
MOTOR ON
```

Turns on the cassette motor.

OPEN Open a Path to a Device

Syntax:

```
OPEN mode,#device,path[,r_size]
```

Function:

Opens a path to a disk file or to a sequential I/O device.

Parameters:

mode A string indicating how the disk file or I/O device will be used.

"I"	Sequential input
"O"	Sequential output
"D"	Direct records
"R"	Random records

device A number to be used in all future references to the disk file or I/O device.

-2	Special -- printer
-1	Special -- cassette
1-15	File or I/O device

path A string giving the name of the file or I/O device to be opened

r_size The size of an individual data record, in bytes. Used only when mode = "D" or mode = "R". The valid range of r_size is 0-32767. The default record size is 256 bytes.

Notes:

Both RSB and DECB allow you to access disk files with the OPEN command. RSB also allows you to open any OS9 SCF-type device for sequential input or output.

Devices identified by negative device numbers are always open. RSB opens the cassette and printer devices automatically at startup. If these devices are not sharable, they caan only be accessed from one RSB window at a time.

The path name can be specified in BASIC format or in OS9 format. If a path name begins with a "/", RSB will interpret it in OS9 format; otherwise BASIC format is used. Refer to Chapter 5 for a description of how directory handles are used when a file name is specified in BASIC format.

Example:

```
OPEN "O",1,"/t2"
```

Opens OS9 device "/t2" as OS9 device 1.

OS9 Perform an OS9 System Call

Syntax:

```
OS9(stat_str)
```

Function:

Performs an OS9 system call with specified register values, and returns the register values after the call.

Parameters:

stat_str	A string or string variable that defines the system call. Must be at lest 10 characters and must have correct format.

Return Value:

String, contains the register values after the system call.

Notes:

The first 10 locations of stat_str are interpreted as follows:

Loations	Function
1	System call code
2	CC register
3-4	D register
5-6	X register
7-8	Y register
9-10	U register

The register values after the call are returned in the same 10 locations.

Stat_str may have up to 255 characters. Any characters after the first 10 may be used as needed; for example, you might set the X register to point at byte 11 and use this portion of the string as a data buffer.

If stat_str is a variable name, the OS9 function will modify the value of that variable when it returns the register values after the system call. If you don't want to change the value of the string variable, use the string '+' operator like this:

```
100 B$=OS9(A$+"")
```

Now the OS9 function will only change B$. Be careful, though, because using the string operator will also change the location of the string in memory.

Example:

```
100 'Print OS9 error message E
110 A$=CHR$(15)+CHR$(0)+CHR$(0)+
    CHR$(E)+"      "
120 A$=OS9(A$):PRINT CHR$(10);
130 RETURN
```

This subroutine calls OS9's PRINTERR routine to display the error message passed as variable 'E'. The PRINT CHR$ (10) was added to force a new line after displaying the message.

PAINT Paint Low-Res Graphics Figure

Syntax:

```
PAINT (x,y)[,p_color,b_color]
```

Function:

On a VDG window, fills an area bounded by a line with a specified color.

On a true window, changes the color of an area to a specified color.

Parameters:

x	Starting X-coordinate, 0-639 or 0-319
y	Starting Y-coordinate, 0-191
p_color	Paint color, 0-15. Defaults to current background color
b_color	Border color, 0-15. Defaults to current foreground color

Notes:

On a true window, RSB's PAINT operates differently than DECB's PAINT. The results of both commands are often equivalent.

Example:

```
PAINT (23,57),3,0
```

Fills the area including point (23,57) and bounded color 0, with color 3.

PCLEAR Allocate Low-Res Screens

Syntax:

PCLEAR count

Function:

Allocates or deallocates low-res graphics screens based on the value of 'count'.

Parameters:

count	The number of 1.5K graphics pages requested.

Notes:

RSB allocates low-res graphics screens from OS9's system memory map. Low-res graphics screens do not use any of RSB's program or variable storage memory.

OS9 low-res graphics screens are always either PMODE 3 or PMODE 4, and use 6K of system memory each. OS9 will only allocate complete screens.

RSB always assumes that the PMODE will be either 3 or 4 when figuring out how many screens to ask for. Any request for a partial screen causes RSB to allocate an entire screen. For example,

PCLEAR 2

will allocate one 6K graphics screen even though there are normally four 1.5K pages in a screen of this size.

PCLEAR commands, in programs that use more than one graphics screen and PMODEs 0-2, may have to be modified to request the correct number of 6K graphics screens.

OS9 allows a maximum of two VDG low-res graphics screens. Depending on the amount of available system memory, OS9 may only be able to allocate one screen.

If your boot file is very large, OS9 may not be able to allocate ANY VDG graphics screens. If there are no low-res screens available, all low-res graphics commands will generate a ?HR ERROR.

The PCLEAR command is ignored on true windows.

Example:

```
PCLEAR 3
```

Reserves one 6K low-res graphics screen.

PCOPY Copy Low-Res Graphics Page

Syntax:

PCOPY source TO dest

Function:

Copies the equivalent of one low-res graphics page to another of the same size.

Parameters:

source Page number to copy from

dest Page number to copy to

Notes:

All RSB low-res graphics screens use 6K of memory, regardless of PMODE. RSB scales the size of a graphics page (normally 1.5K) to match the current PMODE.

For example, a DECB PMODE 0 screen uses one graphics page, or 1.5K, of memory. The same screen uses 6K of memory under RSB, so the RSB PCOPY command will copy 6K of data from one screen to another in PMODE 0. Similary, 3K is copied in PMODE 1 or 2.

PCOPY generates a ?HR ERROR if executed on a true window.

Example:

PCOPY 1 to 2

Copies graphics page 1 to page 2.

PLAY Play Music

Syntax:

PLAY m_info

Function:

Plays music made up of notes and rests described by the string m_info.

Parameters:

m_info	A PLAY parameter string as described in Color Computer 3 Extended BASIC.

Notes:

The PLAY command uses the music device specified in the environment file. If no music device is specified, RSB uses standard output as the music device.

All notes and rests are generated by I$SetStt calls to the music device, with the following register values:

```
Reg-B = SS.Tone
Reg-Y = Pitch Descriptor
        Tone code, 0-4095
Reg-X = Shape Descriptor
        MS byte = amplitude 0-63
        LS byte = number of ticks
```

When standard output is a VDG or true window, tones are generated by the Color Computer 3's built-in interval timer.

The tone code in Reg-Y is related to pitch by a simple formula:

$$f = \frac{1}{B - Ay}; \quad B = 65{,}626\ t; \quad A = 16\ t$$

$$t = 16 * (1/28{,}636{,}363) \quad \text{(bus cycle)}$$

All RSB compatible sound device drivers use the same formulas for pitch and duration.

Example:

```
PLAY "CC#DD#E"
```

Plays the first five half-steps of the musical scale.

PMODE Set Low-Res Graphics Mode

Syntax:

PMODE [mode][,page]

Function:

Selects a low-res graphics mode and graphics screen number.

Parameters:

mode	Determines the effective screen resolution and number of colors for low-res graphics. Must be in the range 0-4. Defaults to the most recently used mode.
page	The page number of a 1.5K graphics page. The low-res screen containing this graphics page becomes the active low-res graphics screen.

Notes:

PMODE and other low-res graphics commands can be used on either VDG windows or true windows. RSB automatically converts low-res graphics commands executed on true windows to the closest equivalent hi-res graphics command. LINE commands are converted to HLINE commands, and so on.

Some low-res graphics commands, like GET, cannot be converted to an equivalent hi-res command. For maximum compatibility with DECB, you should run low-res graphics programs in VDG windows instead of true windows.

All VDG low-res graphics screens use 6K of system memory, regardless of PMODE. RSB allows you to specify graphics modes that require less than 6K, but always converts to the equivalent of PMODE 3 or 4 based on the number of colors needed.

RSB uses HSCREEN 1 hi-res graphics when emulating low-res graphics on true windows.

Example:

```
PMODE 4
```

Select 256 x 192, two-color low-res graphics.

PRINT @ Print at Screen Offset

Syntax:

```
PRINT @offset[print_list]
```

Function:

Move both the text cursor and the print position to the specified offset from the upper left-hand corner of the screen, and execute a PRINT command.

Parameters:

offset	The desired cursor position, given by (32 * row) + column. Must be in the range 0-511.
print_list	A list of printable items, as described in CoCo 3 Extended Color BASIC for the PRINT command. If omitted, only a carriage return is printed.

Notes:

The PRINT @ command works on all 32, 40, and 80 column text screens. Note that only the first 32 columns and the first 16 rows can be accessed on 40 and 80 column screens.

Example:

```
100 WIDTH 32
110 CLS
120 PRINT @269,"HELLO!"
```

Line 100 selects a 32 column text screen.

Line 110 erases the text screen.

Line 120 moves the cursor to near the center of the screen, and prints a text message.

PUT Place Low-Res Graphic Rectangle

Syntax:

```
PUT (sx,sy)-(ex,ey),array[,action]
```

Function:

Copies an array to a rectangular area of the low-res graphics screen.

Parameters:

sx	X-coordinate of one corner of the rectangle, 0-255
sy	Y-coordinate of one corner of the rectangle, 0-191
ex	X-coordinate of the opposite diagonal corner of the rectangle, 0-255
ey	Y-coordinate of the opposite diagonal corner of the rectangle, 0-191
array	The name of a two-dimensional array that stores the graphic rectangle
action	Selects one of several methods to use when placing the rectangle on the screen:

PSET	Place stored image
PRESET	Invert and place
AND	Bit-wise AND
OR	Bit-wise OR
NOT	Invert screen

Notes:

The PUT command generates a ?HR ERROR when used on a true window.

When using GET and PUT together, you must use the same PMODE for both the GET and the PUT.

See also GET.

Example:

```
100 DIM A(19,2)
110 GET (0,0)-(19,2),A
120 PUT (100,100)-(119,102),A
```

Line 100 declares an array big enough to hold a 20 x 3 graphic rectangle. Note that the DIM command reserves elements beginning at subscript 0, so line 100 actually declares a 20 x 3 array.

Line 110 copies the 20 x 3 graphic array described by the diagonal line from (0,0)-(19,2) into array A.

Line 120 duplicates the original rectangle at another screen location.

RENAME Rename a File

Syntax:

RENAME s_name TO d_name

Function:

This command changes the name of a file.

Parameters:

s_name Original file name

d_name New file name

Notes:

RSB calls OS9's RENAME utility to do the actual work.

Examples:

RENAME "TEST.BAS" TO "TEST.OLD"

Changes the name of TEST.BAS on the default drive to TEST.OLD.

SOUND Make a Sound

Syntax:

```
SOUND pitch,length
```

Function:

Generates a tone.

Parameters:

pitch	Specifies the pitch of the tone, 0-255
length	Specifies the duration of the tone, 0-255

Notes:

The pitch parameter is related to the frequency of the sound by a simple formula:

$$f = \frac{1}{B - Ap}; \quad B = 10{,}410\ t; \quad A = 40\ t$$

$$t = 16 * (1/28{,}636{,}363) \quad \text{(bus cycle)}$$

The duration of the sound is specified in 1/15 second increments.

RSB uses SS.TONE service calls to process the SOUND command. Refer to the description of the PLAY command for more information about SS.TONE.

Example:

```
SOUND 159,15
```


Produces a 440 Hz tone for 1 second.

TIMER Elapsed Time

Syntax:

    TIMER

Function:

    Return elapsed time

Return Value:

        number, 0-65535

Notes:

    The TIMER function returns a value that is incremented every 1/60 second.

    RSB obtains the timer value by briefly disabling interrupts, mapping in OS9's system variables, and accessing an area maintained by the OS9 CLOCK module. The returned value is the system timer value minus a programmable offset (see TIMER=).

Example:

```
100 A=TIMER
110 SOUND 100,15
120 PRINT TIMER-A
```

    Line 100 reads the current timer value and stores it in veriable A.

    Line 110 generates a tone that lasts 4*15, or 60, timer increments.

    Line 120 reads the timer value again, calculates the elapsed time, and displays the elapsed time.

The elapsed time in this example will usually just over 60 ticks: the length of of the tone, plus some time for overhead.

TIMER= Calibrate Timer

Syntax:

```
TIMER=time
```

Function:

Calibrate the TIMER funtion.

Parameters:

time	The desired TIMER value, 0-65535.

Notes:

The TIMER= command reads the system timer and calculates an offset that, when subtracted from the timer value, produces the number 'time'. RSB saves this offset and subtracts it from the value of the system timer when processing subsequent uses of the TIMER function.

Example:

```
100 TIMER=0
110 SOUND 100,15
120 PRINT TIMER
```

Line 100 calibrates the value of the TIMER function to zero.

Line 110 generates a tone that lasts 4*15, or 60, timer increments.

Line 120 reads the timer value again, and displays it as the the elapsed time.

The elapsed time in this example will

usually just over 60 ticks: the length of
of the tone, plus some time for overhead.

VERIFY Control Disk Verification

Syntax:

VERIFY {ON|OFF}

Function:

This command is provided for compatibility with DECB. It does not perform any function under RSB.

Examples:

VERIFY OFF

This command is ignored.

WIDTH Set Text Screen Size

Syntax:

WIDTH {32|40|80}

Function:

Set the number of rows and columns on the text screen, and erase the screen.

Parameters:

32	Select 32 columns and 16 rows
40	Select 40 columns and 24 rows
80	select 80 columns and 24 rows

Notes:

The WIDTH command will not change the type of a window. The window remains either a true window or a VDG window.

WIDTH 40 and WIDTH 80 are not allowed on VDG windows.

WIDTH 32, when used from a true window, creates a 32 x 16 text window centered in a 40 x 24 text screen.

Example:

```
WIDTH 32
```

Sets up and clears a 32 x 16 text screen.

Appendix B

Error Messages

RSB Eror Code Summary

Code	Name	Meaning
0	NF	NEXT without FOR
1	SN	Syntax error
2	RG	RETURN without GOSUB
3	OD	Out of DATA
4	FC	Illegal function call
5	OV	Numeric overflow
6	OM	Out of memory
7	UL	No such program line
8	BS	Bad subscript
9	DD	Array redefined
10	/0	Division by zero
11	ID	Illegal in direct mode
12	TM	Type mismatch
13	OS	Out of string space
14	LS	String too long
15	ST	String too complex
16	CN	Can't continue
17	FD	Bad file data
18	AO	File already open
19	DN	Bad device number

Code	Name	Meaning
20	IO	Generic I/O error
21	FM	File mode error
22	NO	File not open
23	IE	End-of-file
24	DS	Not allowed in RUN mode
25	UF	Undefined function
26	NE	No such file
27	BR	Illegal record number
28	DF	Disk full
29	OB	Out of buffer space
30	WP	Write protect
31	FN	Illegal file name
32	FS	Corrupt file system
33	AE	File already exists
34	FO	Field overflow
35	SE	SET without FIELD
36	VF	Write verify failed
37	ER	End-of-record
38	HR	Hi-res graphics error
39	HP	Hi-res text error

OS9 Error Translation

RSB uses the same error codes and error messages as Disk Extended Color BASIC. When RSB receives an error code from OS9, it converts the error code to a standard BASIC code.

OS9 error #002 is converted to a BREAK condition. If your program has executed an ON BRK command, RSB will transfer control to the BREAK service routine whenever it receives error code #002.

OS9 errors lower than E$ICoord (189) are converted to BASIC ?FC ERROR (code 4) errors.

OS9 errors between E$ICoord and E$WUndef are converted to BASIC ?HR ERROR (code 38) errors.

The Table shows how other OS9 error codes are converted into BASIC error codes. Error codes not mentioned above, and not listed in the Table, produce a BASIC ?IO ERROR (code 20) error.

OS9 Error	Code	BASIC Error	Code
E$Poll	196	?OM ERROR	6
E$NoTask	239	?OM ERROR	6
E$NoRAM	237	?OM ERROR	6
E$PrcFul	229	?OM ERROR	6
E$MemFul	207	?OM ERROR	6
E$PthFul	200	?OM ERROR	6
E$DevOvf	204	?OM ERROR	6
E$DirFul	206	?OM ERROR	6
E$NEMod	234	?NE ERROR	26
E$MNF	221	?NE ERROR	26
E$PNNF	216	?NE ERROR	26
E$Seek	247	?FS ERROR	32
E$IBA	219	?FS ERROR	32
E$NES	213	?FS ERROR	32
E$BNam	235	?FN ERROR	31
E$BPNam	215	?FN ERROR	31
E$FNA	214	?FM ERROR	21
E$BMode	203	?FM ERROR	21
E$EOF	211	?IE ERROR	23
E$CEF	218	?AE ERROR	33
E$UnkSVC	208	?FC ERROR	4
E$Unit	240	?DN ERROR	19
E$BPNum	201	?DN ERROR	19
E$WP	242	?WP ERROR	30
E$Full	248	?DF ERROR	28
E$BMHP	236	?FD ERROR	17
E$BMCRC	232	?FD ERROR	17
E$BMID	205	?FD ERROR	17

Appendix C

RSB Environment File

Everybody's Color Computer 3 setup is unique. For example, some people have RGB monitors, while others have composite monitors or television sets.

RSB stores information about your CoCo system in a special environment file. RSB checks this file every time you use the RSB software, in case you've added new equipment or otherwise changed the system.

File Format

The environment file is an ordinary text file. Each line of the file tells RSB something about your system -- such as what type of monitor you have.

Lines that begin with an asterisk (*) are ignored by RSB. You can use this type of line as a comment, to remind yourself of what other lines do.

Blank lines are also ignored. Use blank lines to make the file more readable.

Here's a listing of the standard RSB environment file:


```
*   RSB Environment File

*   Printer device path name (defaults to
*   none).  Use "#" or "/NIL" for default.
/p

*   Cassette path name (defaults to none).
*   Use "#" or "/NIL" for default
#

*   Sound device path name (defaults to
*   standard output).  Use "#" for default
#

*   Monitor type -- RGB or COMPOSITE.
*   Only the first character is significant
COMPOSITE

*   Default screen width.  Must be 32, 40,
*   or 80.  Illegal values default to 80.
*   This value is ignored in VDG windows.
40

*   Right joystick type -- MOUSE or
*   JOYSTICK.  Only the first character
*   is significant
JOYSTICK

*   Default path names for directories
*   0, 1, 2, and 3.  Must be full OS9
*   path names.  These default to "no
*   path".  Use "#" for default
*
/d0
/d1
/d2
/d3

*   End of rsb_env.file
```


Search Path

The RSB environment file is always named rsb_env.file. RSB searches for this file in two different places:

1) /dd/SYS/rsb_env.file
2) /d0/SYS/rsb_env.file

/dd is OS9's "default device"; it may be a floppy disk, hard disk, or RAM disk. RSB checks /dd first, in case your system is using a hard disk or other alternative primary storage. If there is no device named /dd, or there is no RSB environment file on that device, RSB will look for the environment file on device /d0.

RSB should not be used without the RSB environment file.

Appendix D

Programmer's Notes

This Appendix describes some of the inner workings of RSB. This information is provided for the benefit of advanced BASIC or machine language programmers.

Internal Operation

RSB is implemented as an overlay that modifies Disk Extended Color BASIC. The overlay changes all absolute addressing in the code segment into relative addressing, and modifies all of the I/O routines to perform OS9 system calls.

DECB alloates the 32 column text screen at addresses $400-$5FF, and stores several CoCo 3 variables at addresses $FE00-$FEFF. This has been modified in RSB; the text screen is allocated in OS9's system memory, and all references in the range $FE00-$FEFF have been relocated to $500-$5FF. Several new variables have been added in this area.

All other data references use absolute or direct addressing. Level 2 OS9 always allocates RAM to a process from $0000 up, and RSB takes advantage of this when using absolute addressing.

The low-resolution graphics routines use SS.SlGBf and SS.AAGBf calls to allocate

and map VDG graphics buffers. The original DECB routines perform all VDG graphics. Note that the graphics buffers are mapped into unused 8K slots in RSB's memory map, and DO NOT use the RAM allocated to RSB for program and variable storage.

High-resolution graphics use OS9's windowing system for most functions. For example, the HPOINT function sends a GetBlk command to the screen, uses SS.MpGPB to map in the buffer, and then examines the buffer to calculate the value of the graphics point.

The scheme for accessing BASIC's command tables has been modified slightly. DECB for the CoCo 3 uses some very tricky code to add new commands without using the "spare" command table (address $13E-$147). RSB uses the spare command table for these commands.

RSB's new function, OS9(a$), uses token $28. This token was skipped in CoCO 3 DECB due to the tricky command table code.

The path number of each open OS9 device is stored in the associated RSB file control block, at offset FCBDRV ($01). The address of the file control block (FCB) for device #j is stored in the two bytes at location $926 + 2*j. This equation only works for j >= 1.

RSB essentially ignores the four FAT buffers from $800-$927, but portions of the original DECB code may still access this area when setting up FCBs, etc. It's best to LEAVE THIS AREA ALONE and not try to store ML subroutines or other data there.

Memory Map

RSB's memory map is similar to DECB, with several important differences.

The most important difference is that RSB's code segment is fully relocatable; OS9 may load RSB at any address. USUALLY, but not always, this address turns out to be $6000. There are an extra five bytes at the beginning of RSB, so address X in DECB is equivalent to:

```
(X - &H7FFB) + RSB_load_address
```

in RSB. RSB stores its load address in the two bytes at location $5E0.

The second major difference is in how RSB and DECB allocate text / graphics screens. RSB's 32 column text screen is somewhere in OS9's system memory map, and you can't use POKE commands or traditional machine language routines to access it. VDG graphics screens are allocated in unused 8K slots of RSB's memory map, and do not use any of RSB's original memory allocation.

Here's how RSB Version 1.0 uses the block of memory between $400 and $5FF. Future versions of RSB may use this block differently, so USE THESE ADDRESSES AT YOUR OWN RISK!


```
*
*   RSB System Variables
*   Copyright 1988 Burke & Burke
*   All Rights Reserved
*

    org     $400
*
*   The path name variables for drives 0-3
*   All are <CR> terminated.
*
H.DRIVE0    rmb     64
H.DRIVE1    rmb     64
H.DRIVE2    rmb     64
H.DRIVE3    rmb     64

    org     $500
*
*   Super Extended graphics variables
*   Relocated to $500
*
H.CRSLOC    rmb     2
H.CURSX     rmb     1
H.CURSY     rmb     1
H.COLUMN    rmb     1
H.ROW       rmb     1
H.DISPEN    rmb     2
H.CRSATT    rmb     1
            rmb     1
H.FCOLOR    rmb     1
H.BCOLOR    rmb     1
H.ONBRK     rmb     2
H.ONERR     rmb     2

*   $510

H.ERROR     rmb     1
H.ONERRS    rmb     2
H.ERLINE    rmb     2
H.ONBRKS    rmb     2
H.ERRBRK    rmb     1
```

```
H.PCOUNT    rmb     1
H.PBUF      rmb     80

*   $569
*   These are unique to RSB

*   Current palette register settings
H.PALET     rmb     16

*   32 byte "master" GetStt buffer
H.COOKED    rmb     32
*   32 byte "working" GetStt buffer
H.RAW       rmb     32
*   $00 or value of captured signal
H.BREAK     rmb     1

*   $5BA
*   Path table.  An entry of $FF = device
*   closed.  Otherwise, the entry is the
*   number of the open path.  Files are
*   handled by storing the path number in
*   the file control block.

H.Sound     rmb     1   ;Sound I/O path
H.PRNTR     rmb     1   ;Printer I/O path
H.CASSIO    rmb     1   ;Cassette I/O path

*   $00 if composite, $FF if RGB monitor
H.MonTp     rmb     1
*   $00 if right joystick, $FF if mouse
H.JoyTp     rmb     1

*   Window type ($00 = VDG, $FF = window)
H.WTYPE     rmb     1

*   $5C0
*   Table of PD.TYP device type for open
*   "files"
H.DTTBL     rmb     16

*   $5D0
```

```
*   Address of text screen ($00 if none)
H.TSCRN     rmb     2
*   Displayed screen $00 = text
*   $FF = graphics
H.ScTyp     rmb     1

*   Address of graphics screen #0
H.Scrn0     rmb     2
*   Address of graphics screen #1
H.Scrn1     rmb     2

*   Stored OS9 error code (sometimes)
H.OS9ER     rmb     1

*   $FF if '-g' specified, else $00
H.GFLAG     rmb     1

*   Background palette # for 32 column text
H.TBK32     rmb     1
*   Border palette for 40 or 80 column text
H.TXTBD     rmb     1

*   Old foreground color
H.OLDFGC    rmb     1
*   Old background color
H.OLDBGC    rmb     1

*   RSB's Process ID (for graphics buffers)
H.PID       rmb     1
*   PMODE promotion ($FF = emulate VDG)
H.PROMO     rmb     1

*   Stashed device #
H.DEVNUM    rmb     1

*   $5E0
*   Load address of RSB
H.BASE      rmb     2

*   Spare out to $5ED
```



```
*
*    More variables -- uses from here
*    to end of $500 page
*
         org      $5ED

*    $55 if interrupt vectors valid;
*    else $00
INT.FLG      rmb      1

*    end of variables.equ
```

Cassette Format

Cassette I/O is handled by calling an SCF device driver designated in the environment file.

Any device may be designated as the cassette device, but the data stream from that device must use the cassette format.

A full description of the cassette data format is given in Tandy's Color Computer 3 service manual, catalog #26-3334. Here's a summary:

```
FILE =  LEADER (128 bytes $55)
        NAME BLOCK
        LEADER (128 bytes $55)
        DATA BLOCKS
        END OF FILE BLOCK

BLOCK = LEADER BYTE ($55)
        SYNC BYTE ($3C)
        BLOCK TYPE BYTE
            $00=name,
            $01=data
            $FF=end of file
        BLOCK LENGTH ($00-$FF)
        DATA
        CHECKSUM BYTE
            all but leader & sync
        TRAILER BYTE ($55)

DATA (name block only) =
        NAME (8 bytes)
        TYPE
            $00 = BASIC
            $01 = Data
            $02 = ML
        ASCII FLAG
```



```
            $00 = Binary
            $FF = ASCII
        GAP FLAG
            $01 = Continuous
            $FF = Gaps
        ML PROGRAM START ADDRESS
        ML PROGRAM LOAD ADDRESS
```

The length of an end-of-file block is always 0.

Appendix E

OS9 Utilities

HCOPY

Syntax: HCOPY [-option] b_path o_path
HCOPY [-option] o_path b_path

Function: Copy a file between the OS9 and BASIC directories of a SKITZO disk.

Parameters:

option Used to force the type of files written to the BASIC directory.

-0	BASIC program
-1	BASIC data file
-2	BINARY
-3	TEXT file
-4	ASCII BASIC

b_path Identifies the BASIC file name, as in HDEL.

o_path OS9 file path name.

Notes:

HCOPY copies files between BASIC and

OS9 directories.

HCOPY accepts two filenames on the command line. The left-most filename is the source file, and the right-most filename is the destination file.

When copying from BASIC to OS9, HCOPY always creates an OS9 text file.

When copying from OS9 to BASIC, the type of the destination file is determined by the filename suffix, as follows:

.BAS	BASIC program
.DAT	BASIC data file
.BIN	BINARY (machine language)
.TXT	TEXT file
.ASC	ASCII-format BASIC program

Files with no suffix or with suffixes not listed above become BASIC data files.

You can force the type of the file, regardless of suffix, via the -0, -1, -2, -3, and -4 options.

Example:

```
OS9:dir /d0

Directory of /d1   22:39:03

memo.txt

OS9:hdir %d1
HDIR VERSION 2.5
COPR. 1988 BY BURKE & BURKE

Name          Type    F  Size
```

```
------------  ------  -  -------
DISKDUMP.BAS   BASIC  B       96
XTFMT.BIN     BINARY  B     1108
CONFIG.BAS     BASIC  B      154

2304 bytes per granule.
3 granules used.
65 granules available.
149760 bytes available.

OS9:hcopy /d0/memo.txt %d1/NEWMEM.TXT
HCOPY VERSION 2.0
COPR. 1988 BY BURKE & BURKE

OS9:hdir %d1
HDIR VERSION 2.0
COPR. 1988 BY BURKE & BURKE

Name          Type    F  Size
------------  ------  -  -------
DISKDUMP.BAS   BASIC  B       96
XTFMT.BIN     BINARY  B     1108
NEWMEM.TXT      TEXT  A    26224
CONFIG.BAS     BASIC  B      154

2304 bytes per granule.
15 granules used.
53 granules available.
122112 bytes available.

OS9:
```

HDIR

Syntax: HDIR device

Function: Display BASIC directory of SKITZO floppy disk.

Parameters:

device The device name of the floppy drive containing the disk, with a % in place of the leading /.

Notes:

HDIR displays the BASIC directory of a disk that has been processed by the SKITZO command.

The directory shows the name and type of each file, and its size in bytes. The granule size and number of free granules are displayed at the end of the directory.

Example:

```
OS9:hdir %d1
HDIR VERSION 2.5
COPR. 1988 BY BURKE & BURKE

Name          Type    F  Size
------------  ------  -  -------
HYPER2-1.BIN  BINARY  B     2979
HYPERIO.BAS    BASIC  B     1010
```

```
CNFBIN.BIN     BINARY  B      2495
HYPERDRV.BIN   BINARY  B      1002
DISKDUMP.BAS    BASIC  B        96
XTFMT.BIN      BINARY  B      1108
FR.DR          BINARY  B       117
J1.DD          BINARY  B        32
HDFMT.BAS       BASIC  B      3959
CNFTOOL.BAS     BASIC  B     16751
MSATOOL.BAS     BASIC  B      6243
OS9IFY.BAS      BASIC  B      4765
XT.DR          BINARY  B       849
H8.DD          BINARY  B        32
H0.DD          BINARY  B        32
HYPERDEV.BIN   BINARY  B       193
CONFIG.BAS      BASIC  B       154

2304 bytes per granule.
66 granules used.
2 granules available.
4608 bytes available.

OS9:
```


HDEL

Syntax: HDEL b_path

Function: Delete a file from the BASIC directory of a SKITZO disk.

Parameters:

b_path Identifies the file to be deleted. The format of path is

device/file_spec

device is defined as with HDIR. The file_spec is the name of the file to be deleted, in BASIC format. Upper and lower case are distinct.

b_path example:

%d1/TEST.BAS

Notes:

HDEL deletes a file from the BASIC directory of a SKITZO disk.

Any granules used by the file become available to both BASIC and the HCOPY utility.

Example:


```
OS9:hdel %d1/J1.DD
HDEL VERSION 2.5
COPR. 1988 BY BURKE & BURKE

OS9:
```

SKITZO

Syntax: SKITZO device

Function: Divides a newly formatted disk into OS9 and RS-DOS sections.

Parameters:

device The name of the floppy device that holds the disk to be divided.

Notes:

SKITZO works only on 35 track, single sided floppy disks.

Tracks 0-16 are allocated to OS9, and tracks 17-34 are allocated to BASIC.

The OS9 and RS-DOS sections are completely independent. Writing to one section does not alter the other.

The main use of SKITZO is to prepare a disk for subsequent use with HDEL, HDIR, or HCOPY.

Example:

```
OS9:skitzo /d0
```


WIDTH

Syntax: WIDTH {32|40|80}

Function: Change OS9 screen width

Parameters:

32	Select 16 lines, 32 columns
40	Select 24 lines, 40 columns
80	Select 24 lines, 80 columns

Notes:

The WIDTH utility redefines the size of the current window. This command has no effect when used from a VDG window.

WIDTH uses palette register 0 as the foreground color, and palette register 10 as the background color.

The cursor will match palette register 1 when typing new text, and will match palette register 0 when typing over existing text (e.g. due to the <- and CNTL-A keys).

