BASIC09 Programming Language Reference

BASIC09: Programming Language Reference Manual

Revision History

Revision F February 1983

BASIC09: Programming Language Reference Manual

Revision History

Revision H, January 1984

BASIC09: Programming Language Reference Manual

Revision History

Revision A March 2003
Updated for OS-9 Level One Version 02.01.01
Revision by Boisy Pitre

BASIC09: Programming Language Reference

Section of the OS-9 Level II Operating System Manual

Revision History

Printed March, 1988 by Radio Shack/Tandy Corporation

This BASIC09 Programming Language Reference

Built from all the above by L. Curtis Boyle and Wayne Campbell

Revision History

Revision B July, 2018, Revision C, August, 2024
Updated for NitrOS-9 Level One and Two

Contents

Chapter 1

Looking at the Basics

BASIC09 is a computer language created for use with the NitrOS-9 operating system. Along with standard BASIC language statements and functions, it includes the most useful elements of the PASCAL computer language.

In brief, BASIC09's advantages are:

Fast execution speed — BASIC09 compiles procedure lines as you enter them. When you finish a procedure, you can compile it further. The result? Procedures that execute nearly as fast as machine language.

Full feature editing — The text editor features automatic line formatting, search, search and change, global search, global search and change, line renumbering, and much more. You can move in and out of the editor quickly and easily.

Modular programming functions — You can write small, easy-to-under-stand procedures, then chain them to create sophisticated programs. You can call one procedure from another, regardless of whether the called procedure is in memory or on disk. BASIC09/RUNB will handle a maximum of 128 procedures in its process space at once. (You can have more than 128 if you LOAD/KILL them as needed.)

Interfacing to NitrOS-9 — Both you and your procedures can take advantage of almost any NitrOS-9 function from within BASIC09, including the execution of disk management commands and application programs.

Structured programming — You can structure procedures more efficiently and clearly by taking advantage of a variety of loop commands, optional line numbering, and BASIC09's ability to call modules written in other computer languages.

Memory saving — Strings can be any length. For each operation, you can select the most efficient of five available data types.

features	Compiled procedures use less space. You can save several procedures into one file. String variables have a maximum length of 32,767 bytes.
Complex data structures	Combine any type of data into a single dimensioned data structure that you can move, store, and assign easily and quickly.
Sophisticated graphics	BASIC09 has three levels of graphics. The high-resolution graphics and text capabilities feature more than seventy functions.
High speed, precision math	BASIC09 has a full range of fast and accurate math and transcendental capabilities including powers, roots, trigonometry, logic, and boolean functions.
Simple and fast debugging	BASIC09 provides superior debugging functions. It checks syntax as you enter lines. It points to the location of your errors and tells you what they are. You can stop programs, enter the debugger, then continue execution. Execution errors automatically put you in a debugging mode where you can examine values, and step and trace your way through faulty procedures.

Using BASIC09

The Ease-Of-Use (EOU) NitrOS-9 hard drive images include all the Basic09-related modules in the CMDS directory. It is a good idea to make sure you have a copy of this disk image as a backup before you make any changes. (e.g. adding, deleting, or modifying files, creating/deleting directories, etc.)

To use BASIC09, from any command line prompt type:

```
basic09[ENTER]
```

After a short pause, during which NitrOS-9 loads BASIC09 from the drive, the screen displays the copyright notice and a new *prompt*, like this:

```
                BASIC09
          6309 VERSION 01.01.01
COPYRIGHT 1980 BY MOTOROLA INC.
  AND MICROWARE SYSTEMS CORP.
   REPRODUCED UNDER LICENSE
       TO TANDY CORP.
 ALL RIGHTS RESERVED.
```

The B: indicates that your computer is in the BASIC09 *command mode*. From the command mode, you can issue instructions to the system executive to manipulate procedures (programs). If you are using the 6809 version of NitrOS-9, the copyright notice will reflect 6809. Note that running the 6809 version on a 6309 native mode system (like EOU/6309) can cause some errors. So make sure you are using the one that came with your EOU image (don't worry, any programs you write will run fine on both with no changes; it's only BASIC09 and RUNB themselves that need to be the proper version).

Requesting More Memory

Unless you specify otherwise, BASIC09 automatically sets aside 8192 bytes of memory (4096 bytes under Level One) as a workspace into which you can type or load procedures. BASIC09 reserves approximately 1200 bytes (~1.2K) of the workspace for internal use, leaving you with 6992 bytes for workspace (2896 bytes under Level One).

There are two ways to set aside more memory for BASIC09 operations:

- You can reserve extra memory when you first enter BASIC09 by using the NitrOS-9 *memory size* option. For instance, to reserve 18,176 bytes, enter this command to initialize BASIC09:

  ```
  basic09 #18k[ENTER]
  ```

- You can also request additional memory after loading BASIC09. At the command prompt, B:, type:

  ```
  mem 18000[ENTER]
  ```

  This tells BASIC09 to set aside a total of 18,000 bytes of memory, if they are available. (Level One will not have this much available memory.)

In both cases, because BASIC09 rounds the amount you request to the next multiple of 256, the actual reserved memory is 18176 bytes.

> **Note:** If your system does not have enough free memory to reserve the amount you specify, the workspace size does not change.

You can also use the MEM command to reduce memory. However, BASIC09 does not reduce the size of the workspace if doing so destroys resident procedures.

Note: BASIC09 allows a maximum of 40K in its workspace under Level Two (and RUNB allows a maximum of 48K, assuming no external library/procedures are being called as well that at least that aren't pre-merged). Under Level One it will

depend on the size of the OS9Boot file since it shares 64K with the system (unlike Level Two which keeps each process in its own 63.5K workspace).

Writing Procedures

BASIC09 is a *modular* programming language. Several procedures can occupy memory at the same time. Each procedure performs a particular function but can also interact with others to form a sophisticated program.

To create or change procedures, enter the edit mode by typing either `edit[ENTER]` or `[E][ENTER]` at the B: prompt. From now on, when directing you to enter the edit mode, this manual uses the easier to type `[E]` command.

Each time you type a procedure line and press `[ENTER]`, the editor checks for common errors. This automatic checking lets you catch mistakes before you run the program, saving you testing and rewriting time. You can even let the automatic checking help you learn the rules of BASIC09. If you are not sure about a syntax, go ahead and type it the way you think is correct. If you guess wrong, BASIC09 shows where the error is and displays a message to tell what is wrong.

BASIC09's use of modules lets you divide large and complex projects into smaller, easily manageable sections. Not only are the smaller procedures easier to write and understand, they are also easier to test. As well, because BASIC09 lets you *call* procedures that are outside the *workspace* (the computer's memory where you write and edit procedures), you can accumulate libraries of procedures to incorporate into future programs.

You can work on a program's procedures either individually or as a group. For example, to work on the procedures as a group, save your workspace procedures into a single disk file. When you subsequently load the file, BASIC09 automatically loads all of the procedures.

Modules of Other Languages

BASIC09 can incorporate procedures from other languages, such as Pascal, C, or assembly language. Several users can then share the procedures.

Executing Procedures

You execute or *run* procedures from the command mode. When you enter a procedure, BASIC09 compiles it. This means that a procedure is ready for execution as soon as you exit the edit mode. For instance, if you create a procedure named Greeting, you can execute it by typing from the command mode:

```
run greeting[ENTER]
```

Leaving BASIC09

There are two ways you can exit from BASIC09:

- At the B: prompt, type:

  ```
  bye[ENTER]
  ```
- Or, at the B: prompt, press `[CTRL][BREAK]`.

When you use either method, the NitrOS-9 prompt appears immediately indicating that the operating system is waiting for a command.

> **Note:** When you exit BASIC09, you lose all procedures residing in the workspace. Be sure to save them on disk before leaving BASIC09.

The Keyboard and BASIC09

You can use some keys and *key sequences* to produce special characters and to accomplish special BASIC09 functions. You initiate a key sequence by pressing one key and holding it down while pressing a second key. The following list summarizes your keyboard's special functions:

[ALT] Produces graphic characters. Press [ALT] *char* where *char* is a keyboard character.

[CTRL] A control key that you use with other keys. (See

	below.)
[BREAK] or **[CTRL][E]**	Stops the current program execution and returns to the B: prompt in BASIC09's command mode.
[LEFT ARROW] or **[CTRL][H]**	Moves the cursor back one space.
[RIGHT ARROW]	Moves the cursor right one space.
[CTRL][_] or **[CTRL][-]**	Produces an underscore character.
[CTRL][,]	Produces a left brace ({).
[CTRL][.]	Produces a right brace (}).
[CTRL][#]	Produces a tilde (~).
[CTRL][/]	Produces a backslash (\).
[CTRL][BREAK]	Performs an ESCAPE function and sends an end-of-file message to a program receiving keyboard input. To be recognized, [CTRL][BREAK] must be the first thing typed on a line.
[SHIFT][BREAK]	Stops execution of a program and causes BASIC09 to enter the Debug mode.
[CLEAR]	Displays the next window.
[SHIFT][CLEAR]	Displays the previous window.
[CTRL][0]	Activates or deactivates the shift lock function.
[CTRL][1]	Produces a vertical bar (|).
[CTRL][7]	Produces an up arrow or caret (^).
[CTRL][8]	Produces a left bracket ([).
[CTRL][9]	Produces a right bracket (]).

[SHIFT][RIGHT ARROW]	Redisplays the last line typed, and positions the cursor at the end of the line, but does not process the line. Press [ENTER] to process the line, or edit the line by backspacing. If you edit, press [SHIFT][RIGHT ARROW] again to display the edited line.
[SHIFT][LEFT ARROW] or **[CTRL][X]**	Move to beginning of line.
[CTRL][RIGHT ARROW]	Insert character under cursor.
[CTRL][LEFT ARROW]	Delete character under cursor.
[SHIFT][RIGHT ARROW]	Redisplays the current command line.
[CTRL][W]	Temporarily halts video output. Press the spacebar to resume output.
[ENTER]	Performs a carriage return or executes the current command line.

Chapter 2

Sample Session

Although BASIC09 has several functions or modes, they all work together to make programming as simple as possible. The easiest way to learn how BASIC09 and its functions operate is to write and run a program. This chapter provides sample statements and instructions to help you learn how to use BASIC09.

To create and execute a program:

1. Load BASIC09 and enter the edit mode.
2. Type the BASIC program.
3. Enter the system mode and test the program's execution.
4. Debug the program. (Correct any programming errors.)
5. Save the completed program on disk.
6. Load the program into memory and use it.

To begin the program, execute BASIC09. To be sure you have enough room in which to work, reserve a workspace of 10,000 bytes by typing:

```
basic09 #10k[ENTER]
```

The BASIC09 system mode prompt, B:, appears after the copyright message. In the system mode, you can do such things as save and load procedures, change workspace size, and rename and delete procedures.

Creating a Procedure

To write procedures, you must be in the edit mode. You get there by typing:

```
e[ENTER]
```

This causes the screen prompt to change to E:, and the screen displays:

```
PROCEDURE Program
```

Because you didn't give a program name when you entered e, BASIC09 selects the name Program for you. Now, you must write the code to make Program do something.

Commands and Statements

There are two responses you can give at the edit mode prompt. You can type an edit command, or you can type a *statement*. A statement is a text version of an instruction that BASIC09 will execute. It consists of a *keyword*, an *expression*, or a combination of *keywords* and *expressions*. If you type a statement, **you must type a space as the first character of the line.** If you type an edit command, do **not** precede it with a space. To make listings easier to read, this manual uses the symbol ␣ to indicate spaces before every statement. It also uses the ␣ symbol in some statements to indicate the correct number of spaces needed. Whenever you see a space or a ␣ symbol in a procedure you are typing, press the spacebar.

NOTE: You may want to edit your source code files using one of the text editors found on the NitrOS-9 EOU hard disk. You can run the editor in a different window, editing your source as needed, then save the changes, followed by reloading the source file into BASIC09.

To type the procedure in this chapter, begin each line at the E: prompt. After typing a line, check it for mistakes. If you make a mistake, use `[LEFT ARROW]` to move the cursor back. Correct the mistake. Then, type the remaining portion of the line. If there are no mistakes, press `[ENTER]`.

BASIC09 checks each line when you press `[ENTER]`. If you make a mistake in syntax or form, BASIC09 displays an error message. An arrow points to the place in the program line where the error occurred, and a message number indicates the type of error. Refer to Appendix A for an explanation of the error codes.

If, after you enter a line, you find that you made a mistake, type `d[ENTER]` to delete the line. Then, retype the line. Later, the manual tells you how to change text in existing lines.

The following program helps you do a bit of arithmetic. To get a feel for BASIC09, type and execute the program as directed. Remember, when you see a space or ␣, press the spacebar.

```
␣DIM NUMBER1, NUMBER2:INTEGER
␣INPUT "Type a number... "; NUMBER1
␣INPUT "Type another... "; NUMBER2
␣PRINT "The sum of the numbers is ... ";
␣PRINT NUMBER1 + NUMBER2
␣END
```

Executing a Procedure

To execute the procedure, quit edit mode by typing q `[ENTER]`. The compiler further processes your procedure, and the B: prompt reappears. To execute the procedure type:

```
run[ENTER]
```

Type in numbers when asked, and the procedure produces their sum. If you want to save the program on disk, the next chapter tells you how. Chapter 4 introduces several other edit mode commands to search, display, insert, and change statements and text.

Chapter 3

The System Mode

At the B: prompt, you can enter system commands in either upper- or lowercase letters. Some commands operate on the procedures in the workspace. Others provide functions independent of any procedures. Following is a list of all system commands and their purposes.

Command	Function
$	Calls the shell command interpreter to execute a NitrOS-9 command.
BYE or **[CTRL] [BREAK]**	Returns you to the NitrOS-9 system or to the program that called BASIC09. **NOTE:** Any procedures that you have edited will be ***abandoned***. If you wish to save your work, make sure to **SAVE** your procedures first!
CHD	Changes the current NitrOS-9 data directory.
CHX	Changes the current NitrOS-9 execution directory.
[ENTER] or DIR	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or E	Enters the procedure editor-compiler mode.
KILL	Erases one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a disk file into the workspace.
MEM	Displays in bytes the current workspace size, or reserves a specified amount of memory for the workspace.
PACK	Condenses (compiles) one or more procedures. **NOTE:** Unless you **SAVE** the procedures before PACKing, you will lose the changes you made (or the entire source if you haven't saved it before)!
RENAME	Changes a procedure’s name.

RUN	Causes a procedure in the workspace to execute.
SAVE	Writes one or more procedures to disk.

Renaming Procedures

BASIC09's RENAME function is important for two reasons: First, it lets you load into the workspace procedures that have the same name. After you rename the workspace procedure you can load the second file. Second, if you let BASIC09 use the default procedure name, "Program", you can rename the procedure before saving it to disk. By doing this you avoid writing over—and destroying—an existing procedure file. When naming procedures, do not use extension names (e.g. *.prog*) as BASIC09 will strip it off if you try to LOAD a procedure with a "**.**" in the module name. Using extension names in the files you save (such as *.b09*) are acceptable.

To change the name of the procedure you created in the previous program from "Program" to "add", type:

```
rename program add[ENTER]
```

Listing Procedure Names

You can use the DIR command to see if RENAME worked properly. DIR list the names and sizes of all procedures in memory. Because programmers use this command frequently, the system recognizes a shorthand call. Instead of typing `dir[ENTER]` you only need to press `[ENTER]`. This displays a table of the procedures in the following format:

```
Name        Proc-Size     Data-Size
*add        182           32
 add1       217           42
 add2       218           42
2198 free
```

Proc-Size refers to the number of memory bytes required for the procedure. *Data-Size* refers to the number of memory bytes required for the procedure's variables and data structures. The asterisk indicates the current procedure. System commands act on the current procedure unless you indicate otherwise.

The last line of the DIR display tells you how many free bytes of memory remain in the BASIC09 workspace.

Listing Procedures

You can use the LIST command to view procedure lines. To display the current procedure, type:

```
list[ENTER]
```

For example, this is a listing for a procedure named Alpha:

```
PROCEDURE Alpha
0000     DIM A:STRING
0007     DIM T:INTEGER
000E
000F     PRINT "Here is the alphabet backwards:"
0032     PRINT
0034     FOR T=90 TO 65 STEP -1
004A     PRINT CHR$(T);
0051     PRINT " ";
0057     NEXT T
0062     PRINT
0064     PRINT
0066     END
```

You can list multiple procedures to the screen using the following methods:

`list*[ENTER]` Lists all procedures in the workspace to the screen

`list proc1,proc2,proc3[ENTER]` Lists proc1, proc2, and proc3 to the screen

Each procedure line is known as an instruction. When you list a BASIC09 procedure, the system precedes each instruction with a *relative storage address*. The relative address of the first instruction is always 0, and this marks the beginning of the procedure. In the original BASIC09 reference, the memory allocation definition for each instruction is *storage units*. One storage unit in BASIC09 is a *byte*. In the previous example, the second instruction begins 8 bytes from the beginning, and the first instruction is 7 bytes long (00-06 hexadecimal). The third instruction begins 15 bytes from the beginning and shows the second instruction is also 7 bytes long (07-0D hexadecimal). Upon entering an instruction, BASIC09 compiles it, and the length of the source text does not reflect the size of the instruction.

When an error occurs, these addresses provide a way for the compiler to let you know where the error occurred. They are known as I-Code, or Intermediate Code, addresses.

Because BASIC09 compiles programs into I-Code, it must *disassemble* them before it can display them on the screen. This means the lines might not look exactly as typed. For instance, BASIC09 converts lowercase *keywords* (command names) to uppercase. BASIC09 also eliminates some spaces. If your program uses control statements, such as IF/THEN, FOR/NEXT or LOOP/ENDLOOP, the lines in these decision making or looping structures are indented as shown in the Alpha example. Regardless of the appearance of your listed procedures, they execute correctly if you type their commands correctly.

Listing Procedures to a File

There might be times when you want to send a formatted procedure listing, including I-Code addresses, directly to a file. You can do this using NitrOS-9's redirection symbol, >. To save the Alpha procedure to a file named Alpha.list in the current data directory, type:

```
list alpha >alpha.list[ENTER]
```

If you have several procedures in the workspace and want to list more than one to a disk file, separate the procedure names with commas, like this:

```
list alpha_one,alpha_two,alpha_three >
alpha.all[ENTER]
```

In both of the preceding cases, the system creates the Alpha.list and Alpha.all files and stores the specified listings in them. If you use a filename that already exists, BASIC09 displays the prompt:

```
Rewrite?:
```

If you press `[Y]`, the system destroys the original file and overwrites it with the new listing. If you press `[N]` the LIST process terminates.

If you wish to list a procedure, or a group of procedures, to a file that is not in the current data directory, be sure to specify the complete pathlist, such as:

```
list alpha > /d1/programs/alpha.rev[ENTER]
```

Listing Procedures to a Printer

In the same manner as you list procedures to a disk file, you can list one or more procedures to your printer. Make certain your printer is connected and turned on, then again use the redirection symbol, but this time specify the printer device, like this:

```
list alpha > /p[ENTER]
```

Or:

```
list alpha_one,alpha_two,alpha_three > /p[ENTER]
```

Using a Wildcard

Using the NitrOS-9 wildcard, `*`, you can list all procedures in the workspace. For instance, if the procedures Alpha_one, Alpha_two, and Alpha_three exist, list them to the screen by typing:

```
list*[ENTER]
```

Send the list to a file by typing:

```
list* alpha.all[ENTER]
```

Or send the list to your printer by typing:

```
list* /p[ENTER]
```

Note: When you use the wildcard, the name of the file or device to receive the listing immediately follows the LIST* command. Do not use the redirection symbol.

Saving Procedures

You can save one or more procedures to disk using the SAVE command. Unlike LIST, SAVE does not include relative addresses. However, the syntaxes for the LIST and SAVE commands are identical. To save the procedure Alpha to the current data directory, type:

```
save alpha alpha.bak[ENTER]
```

If Alpha is the current procedure, you can save it in a file named Alpha by typing `save[ENTER]`.

To save all of the procedures in the workspace to a file named All.programs in the current data directory, type:

```
save* All.programs[ENTER]
```

As with LIST, to save one or more procedures in a file that is not in the current data directory, make sure you specify a complete pathlist.

To save all of the files in the workspace to a disk file with the same name as the current procedure, type `save*[ENTER]`.

If the disk file you specify does not exist, BASIC09 creates it. If it does exist, the system displays the prompt:

```
Rewrite?:
```

Press `[Y]` to write over the old file with the specified file. The old file is destroyed.

Press `[N]` to terminate the SAVE operation.

Loading Procedures

To load a saved procedure back into BASIC09's workspace, use the LOAD command and specify the appropriate pathlist. For instance, if your current directory is still the directory containing Alpha.bak, load the procedure by typing:

```
load alpha.bak[ENTER]
```

To load Alpha.bak from the PROGRAMS directory on Drive /D1, type:

```
load /d1/programs/alpha.bak[ENTER]
```

You can run and edit a loaded procedure in exactly the same manner as you would a procedure you created.

You can load any number of procedures into the workspace (up to a maximum of 128) as long as your computer has sufficient memory. However, be careful that you do not load a procedure with the same name as a procedure already existing in the workspace. If you do, the new procedure overwrites (destroys) the original procedure. You can rename workspace procedures to avoid this problem.

Deleting Procedures from the Workspace

You can clear the workspace of one or more procedures using the KILL command. For instance, to remove `Alpha` from the workspace, type:

```
kill alpha[ENTER]
```

To remove more than one procedure from the workspace, separate the procedure names with commas. To delete Alpha_1 and Alpha_2, type:

```
kill alpha_1,alpha_2[ENTER]
```

To clear the entire workspace, regardless of the number of procedures it contains, use the BASIC09 wildcard, *. Type:

```
kill*[ENTER]
```

Changing Directories

You change working directories in BASIC09 and NitrOS-9 in the same manner, by using the CHD and CHX commands. CHD changes the working directory and CHX changes the execution directory.

BASIC09 saves files in, or loads files from, the data directory, unless you specify differently in the command pathlist. It stores PACKed procedures in, or loads PACKed procedures from, the execution directory, unless you specify differently in the command pathlist.

Also, if you want to access NitrOS-9 commands from BASIC09, the system first looks for the commands in memory. If they are not there, it looks for them in the execution directory, unless you specify differently.

If your data directory is the ROOT directory, and you wish to change to a directory named PROGRAMS that is a subdirectory of the ROOT directory, type the following command from the command mode B: prompt:

```
chd programs[ENTER]
```

If your current execution directory is the system's CMDS directory, and you want to change to a CMDS directory in the subdirectory BASIC, type:

```
chx basic/cmds[ENTER]
```

Whenever you change to a directory other than an immediate subdirectory, specify a complete pathlist.

Executing NitrOS-9 Commands

BASIC09 lets you use NitrOS-9 commands at any time from the system mode. To do so, precede the command with a dollar sign (`$`). For instance, to look at the current data directory, type:

```
$dir[ENTER]
```

To view the current execution directory, type:

```
$dir -x[ENTER]
```

All NitrOS-9 commands are available, and you can copy files, format diskettes, list files, or use any other functions from the system mode. The only restriction is that your computer must have enough free memory to handle the command you call. If you find that there is not enough memory, try using the MEM command to reduce reserved memory. Then, try the command again.

If you plan on executing multiple NitrOS-9 commands in a row, you can type '$' by itself to spawn a child shell, where you can run whichever commands you need to. To return to BASIC09, type 'ex' at the Shell prompt.

CAUTION: Take care when using NitrOS9 commands run from BASIC09. Some commands will change the screen mode, colors, etc. that may make BASIC09 unreadable when the called command exits. Generic command line style commands should be fine.

Auto-Execute Procedures

The BASIC09 compiler makes two passes through the procedures you write. When you edit the procedure, the compiler performs an initial compilation, checking for any syntax errors. When you leave the edit mode, the system compiles the procedure a second time and checks for any programming errors (such as incomplete loop structures, etc.). With the PACK command, you can further compile your procedures so that they are smaller and execute even faster.

> **NOTE:** Redirection does not work when using pack* (packing all procedures).

PACK causes an extra compiler pass that removes names, line numbers, and non-executable statements. **Before packing a procedure, be sure you save it. Unless you do so, you cannot make further changes to the procedure.**

Once you pack a file, you cannot list or edit the packed version. However, if you save the procedure to disk before packing, you can still list and edit the original file, then pack it again.

When you save a packed procedure on disk, BASIC09 does not normally store it in the data directory. Because the procedure is now *executable*, the system stores it in the current execution directory.

For instance, to convert `Alpha` to a packed procedure in the execution directory, type:

```
pack Alpha[ENTER]
```

If you want to save a packed procedure under a different filename, use the NitrOS-9 redirection symbol:

```
pack Alpha > backwards[ENTER]
```

After packing a procedure, you can delete it from the workspace. If you then run it, BASIC09 automatically loads the file from disk and executes it.

The following is a sequence of commands that demonstrate packing and executing a procedure named Alpha.

`pack Alpha[ENTER]`	packs the procedure and stores it in the execution directory.
`kill Alpha[ENTER]`	deletes the procedure from the workspace.
`run Alpha[ENTER]`	loads the file into memory outside the workspace and executes it.
`kill Alpha[ENTER]`	deletes the module from memory.

You do not need to kill the file immediately after execution, but until you do, the file reduces available memory.

Chapter 4

The Edit Mode

You briefly used the BASIC09 built-in editor to create the Add procedure in Chapter 2. In addition to the features you learned there, the editor has other important functions.

Although you can use any text editor or word processor to write BASIC09 procedures, the BASIC09 editor offers two handy features:

- It is both *string* and *line number* oriented. You can search for strings of characters, and replace them, and you can reference text with optional line numbers.
- It interfaces with the compiler and *decompiler*. This feature lets BASIC09 check continuously for syntax errors and enables you to use procedures that conserve memory.

Edit Commands

Command	Function
`[ENTER]`	Moves the edit pointer to the next line. Causes a command to execute.
+number	Moves the edit pointer ahead ***number*** lines.
+*	Moves the edit pointer to the last line.
-number	Moves the edit pointer back ***number*** lines.
-*	Moves the edit pointer to the first line.
text	Inserts an unnumbered text line before the current line.
ntext	Inserts a line numbered ***n*** in its correct numbered position.
n	Moves the edit pointer to the line numbered ***n***.
Command	**Function**

c/str1/str2	Changes the next occurrence of ***str1*** to ***str2*** from the current line.
c*/str1/str2	Changes all occurrences of ***str1*** to ***str2*** in the current procedure from the current line until the end of the procedure. To change all occurrences in the procedure, use − * to move the edit pointer to the top of the procedure.
d	Deletes the current line.
d*	Deletes all the lines in the procedure.
l	Lists the current procedure line.
l n	List **n** lines from the current position.
l*	Lists all the procedure lines.
q	Terminates the edit session.
r	Renumbers lines beginning at the current line in increments of 10.
r*	Renumbers all lines in increments of 10.
r n	Renumbers lines beginning at line ***n*** in increments of 10
r n1 n2	Renumbers lines beginning at line ***n1*** in increments of ***n2***.
s/string/	Searches for the next occurrence of ***string*** from the current line.
s*/string/	Searches for all occurrences of ***string*** in the current procedure from the current line until the end of the procedure. To search all occurrences in the procedure, use − * to move the edit pointer to the top of the procedure.

Using the Editor

The easiest way to understand the edit commands is to use them. The following sections show you the functions of BASIC09 edit mode.

The manual uses line numbers in the following procedure to acquaint you with all of the functions of the editor. Remember, however, that line numbers are not required with BASIC09. Procedures and programs without line numbers are shorter, faster and easier to read.

First, you need a procedure to work with. Position yourself in the system mode. Then, type this line:

```
e Prose[ENTER]
```

Now, type the following. (Remember, the □ represents a space.)

```
□100 DIM PHRASES(30):STRING
□120 FOR T=1 TO 30
□130 READ PHRASES(T)
□140 NEXT T
□160 PRINT
□170 FIRST=RND(10)
□180 SECOND=RND(9)+11
□190 THIRD=RND(9)+21
□200 PRINT PHRASES(FIRST);
□210 PRINT PHRASES(SECOND);
□220 PRINT PHRASES(THIRD);
□240 PRINT
□300 DATA "Love□","An orange□","Humanity□","A
kiss□"
□310 DATA "A dark cloud□","A goose feather□","A
popsicle□"
□320 DATA "Home cooking□","Cold pizza□","Rock n'
Roll□"
□330 DATA "is charming like□","makes me dream
of□"
□340 DATA "is as sticky as□","can ooze
like□","smells like□"
□350 DATA "can be as tough to forget as□","can
hurt like□"
□360 DATA "can be as cynical as□","makes a mockery
of□"
□370 DATA "drives me as crazy as□"
□380 DATA "a sticky lollipop.","a web of
intrigue."
```

```
390 DATA "castor oil.","a chocolate bath.","a
broken toe."
400 DATA "honey and things.","personal
defeat.","a wet diaper."
410 DATA "strange happenings.","a pennyless
purse."
```

When you finish typing the procedure, type `q[ENTER]` to return to the system mode. Now you can test the program by typing either:

```
run[ENTER]
```

or

```
run prose[ENTER]
```

After trying the procedure, return to the edit mode by typing `e[ENTER]`.

After displaying the procedure's name, the editor displays line 100 preceded by an asterisk. The asterisk lets you know which line is the *current line* (or the line at which the edit pointer is located.)

Searching Through a Procedure

You can examine a procedure in three ways:

- Press `[ENTER]` to display the procedure one line at a time.
- Skip through the procedure to a particular line.
- List part or all of the procedure to the screen.

When you use either of the first two methods, the line you select to display becomes your current line. When you use the third method, the current line does not change.

Using [ENTER]

If you are still positioned at line 100, but want to examine the first line of data, line 300, press `[ENTER]` 12 times to move down.

Using the Plus Sign to Move Forward

Another method of moving to a specific line is to type a plus sign followed by the number of lines you need to advance to get there. Positioned at line 100, you can type:

```
+12[ENTER]
```

Whether you press [ENTER] or use the plus sign, the last line displayed is now your current line.

Accessing a Line Using the Line Number

The third way to move to a particular line is to type the line number, followed by [ENTER]. For instance, to jump back to line 100, type:

```
100[ENTER]
```

The editor displays line 100 and makes it your current line.

Using the Minus Sign to Move Backward

In the same manner that you move forward in the procedure using the plus sign, you can move backward using the minus sign, or hyphen.

Type 300[ENTER] to return to line 300. To display line 240 and make it your current line, type:

```
-[ENTER]
```

To display line 190 and make it your current line, type:

```
-4[ENTER]
```

The Global Symbol

The BASIC09 editor also makes use of the asterisk as a global symbol. For instance, following a command with an asterisk causes that command to affect the entire procedure.

This feature allows you to move quickly to the beginning and end of the procedure. To return to line l00, the first line, type:

```
-*[ENTER]
```

To move to the end of the procedure, past all the numbered lines, type:

```
+*[ENTER]
```

Using List

The LIST command lets you select one or more lines for display on your screen. To see this, make the first line your current line, then type:

```
l[ENTER]
```

To list one or more lines, type the LIST command followed by the number of lines you want displayed. For instance, typing l5[ENTER] causes the current line and

four others to appear on the screen, as shown in the following sequence of commands and the resulting display:

```
-*[ENTER]
l5[ENTER]
0000 100  DIM PHRASES(30):STRING
000F 120  FOR T=1 TO 30
0024 130    READ PHRASES(T)
0031 140  NEXT T
003F 160  PRINT
```

You can also use LIST with the BASIC09 global symbol, `*`. Typing an asterisk after the LIST command produces a listing of the entire procedure.

Deleting Lines

Earlier, the manual showed that you can delete the current line by typing `d[ENTER]`. Because this is such a simple process, be sure you don't do it by accident. Removing the wrong line, or too many lines, is very frustrating in a complex procedure.

You can also remove a group of lines from a procedure by typing `d`, followed by the number of lines you want to delete. This command deletes the current line and specified following lines. Again, be careful.

You can remove all of the lines in a procedure by using the global symbol, `*`. Typing `d*[ENTER]` erases all procedure text. However, the procedure name still resides in the workspace. To delete an entire procedure, including the name, use the KILL command from the system mode.

If you decide you don't like the nouns used in the DATA lines of the Prose procedure, erase all the DATA lines containing nouns (lines 300-320) and replace them. To do so, make line 300 your current line by typing:

```
300[ENTER]
```

Then type:

```
d[ENTER]
```

Line 300 disappears and line 310 takes its place as the current line. To finish deleting lines 310 and 320, simply type d[ENTER] two more times.

Alternatively, another method of deleting the DATA lines 300-320 uses only one command. To delete lines 300 through 320, follow the DELETE command with the number of lines you want to remove—in this case, three:

```
d3[ENTER]
```

Lines 300, 310, and 320 disappear. Line 330 becomes the current line. Move back a line to make sure the deletions worked. The line numbers now skip from 240 to 330.

Now you need new nouns for the procedure. Type them in the same style as the old lines, such as:

```
␣300 DATA "A telephone␣","A tickle␣","A girl␣","A boy␣"
␣310 DATA "Bad luck␣","Money␣","A bad bet␣","A lumpy bed␣"
␣320 DATA "A deep thought␣","Sunlight␣"
```

Save a copy of your procedure to disk by exiting the editor and using the SAVE command. Then return to the edit mode and try the global delete by typing:

```
d*[ENTER]
```

Changing Text

Using CHANGE tells the editor to search for existing text and replace it with new text. CHANGE, like DELETE, can easily cause unwanted results if you are not careful. CHANGE will change the desired text from the current instruction. CHANGE-all will change from the current instruction to the end of the procedure being edited.

The CHANGE command requires that you use *delimiters* to separate the command from the search text, and to separate the search text from the new text. You can select from any of the following characters for a delimiter, as long as it does not appear in either the search text or the new text:

```
!#%^&()-+={}[]"'<>,.?/\|
```

Do not use the global symbol (`*`) for search and replace operations. This manual uses a (`/`) as the CHANGE delimiter.

The following steps outline the correct use of CHANGE:

1. Position the editor either before or on the line in which you want to make a change.
2. Type `c` (for CHANGE). Do not use a preceding space.

3. Type a delimiter character, such as /.
4. Type the characters to be changed, following them with the delimiter.
5. Type the new text, followed by the delimiter, closing delimiter optional. (Do not confuse this with the NitrOS-9 edit command, where the closing delimiter is required.)
6. Press [ENTER].

Note: It is a good idea to turn on NitrOS-9's upper- and lowercase function before attempting change or search operations. If you do not, you cannot tell whether the text you want to find is upper- or lowercase, or some combination of the two. If you type the wrong case, the change or search fails.

In case you didn't notice when typing the procedure, line 410 contains an incorrectly spelled word, pennyless. To correct this error, type the following:

```
c/pennyless/penniless/[ENTER]
```

Immediately, the editor displays line 410, with pennyless changed to penniless.

Suppose you decide to change the number of sentence combinations available in Prose. The procedure now has 30 data entries. If you add five subjects, five verb phrases, and five objects, the procedure also needs other changes (for instance, the DIM statement in line 100, the loop size in line 120, and the RND statements in lines 170 through 190).

A quick way to change the number 30 in lines 100 and 120 is to use CHANGE's global function. To change all occurrences of 30 to 45, position the editor at line 100, and type:

```
C*/30/45/[ENTER]
```

Use the CHANGE and global CHANGE functions to adjust the RND statements in lines 170, 180, and 190.

As well as making changes, you can use the CHANGE command to quickly delete portions of text within a line. To do this, type delimiters without new text, in this fashion:

```
C/ feather//[ENTER]
```

This command changes the text "`A goose feather`" in line 310 to "`A goose`". The closing delimiter in this instance is not optional.

Searching for Text

The editor's SEARCH command, S, works in the same manner as the CHANGE command. However, SEARCH only requires you specify a block of text to find. SEARCH will search for the desired text from the current instruction. SEARCH-all will search from the current instruction to the end of the procedure being edited.

With SEARCH, you use delimiters to enclose the text to find. To test the function, position the editor at the beginning of text by typing:

```
-*[ENTER]
```

Now, search for the word `phrases`, by typing:

```
s/phrases/[ENTER]
```
(closing delimiter optional)

The screen displays:

```
*0000 100 DIM phrases(30):STRING
```

To find all occurrences of `phrases` throughout the procedure, use the global symbol. Type:

```
s*/phrases/[ENTER]
```

Renumbering Lines

The RENUMBER command, `R`, renumbers all numbered lines and all references to numbered lines. You can give RENUMBER either one or two parameters. The first is the beginning line number. The second is the increment you want. The default increment is 10.

For instance, the Prose procedure line numbers skip from line 100 to line 120. You can renumber the entire procedure by moving the editor to line 100, and then typing:

```
r 10[ENTER]
```

To change the numbering to increments of 5, beginning at line 100, type:

```
r 100,5[ENTER]
```

You can also change line numbering in portions of the procedure. To do this move the editor to the line where you want the new numbering to begin. Then, type in the new parameters. To renumber line 100 as line 200 and continue with increments of 10, position the editor at line 100. Then type:

```
r 200,10[ENTER]
```

If you are not positioned at the first line of the a procedure, but you wish to renumber all lines, you can use the global symbol to do the job. From anywhere in the procedure, type:

```
r* 100,10[ENTER]
```

This renumbers the entire procedure and increments of 10.

Adding Lines

There are two ways to add new lines to a procedure. You can:

- Position the editor one line below the position for the new line. Then type the new line and press `[ENTER]`. When inserting lines without numbers, Be sure to type a space as the first character of the line tell the editor that the following text is a new procedure line.
- Type a new line, giving it a line number that falls between two existing line numbers.

The following procedure adds more choices to the Prose program. It also adds a new feature that lets you press `[ENTER]` for additional output, rather than having to rerun the procedure. Use the information presented in this section to help you insert the new lines into your program. Because you must change some lines, as well as add lines, the following listing includes the entire procedure.

Referring to the original Prose listing, the lines to change are: 100, 120, 170, 180, and 190.

The lines to add are: 110, 150, 230, 250, 260, 270, 305, 325, 372, 374, 376, 420, 430.

```
PROCEDURE Prose
100 DIM PHRASES(45):STRING
110 DIM RESPONSE:STRING
120 FOR T=1 TO 45
130 READ PHRASES(T)
140 NEXT T
150 REPEAT
160 PRINT
170 FIRST=RND(15)
180 SECOND=RND(14)+16
190 THIRD=RND(14)+31
200 PRINT PHRASES(FIRST);
210 PRINT PHRASES(SECOND);
```

```
220 PRINT PHRASES(THIRD);
230 PRINT
240 PRINT
250 PRINT "□□□□□□□□□□Press ENTER for another
witticism…"
260 INPUT "□□□□□□□□□□Or press the SPACEBAR
and press ENTER to end…",RESPONSE
270 UNTIL RESPONSE>""
300 DATA "Love□","An orange□","Humanity□","A
kiss□"
305 DATA "A computer□","A book□","Misery□"
310 DATA "A dark cloud□","A goose feather□","A
popsicle□"
320 DATA "Home cooking□","Cold pizza□","Rock n'
Roll□"
325 DATA "Snow in June□","A glass house□"
330 DATA "is charming like□","makes me dream of□"
340 DATA "is as sticky as□","can ooze
like□","smells like□"
350 DATA "can be as tough to forget as□","can hurt
like□"
360 DATA "can be as cynical as□","makes a mockery
of□"
370 DATA "drives me as crazy as□"
372 DATA "can bother me like□","blackens my hopes
like□"
374 DATA "can tickle me like□","can be as funny
as□"
376 DATA "has the effect of□"
380 DATA "a sticky lollipop.","a web of intrigue."
390 DATA "castor oil.","a chocolate bath.","a
broken toe."
400 DATA "honey and things.","personal defeat.","a
wet diaper."
410 DATA "strange happenings.","a penniless purse."
420 DATA "a slimy snake.","personal defeat.","a bad
habit."
430 DATA "a bad memory chip.","a good fight.","a
silly friend."
```

The Next Step

Even the best programmers make mistakes—a lot of them. BASIC09 provides a way to catch programming mistakes quickly and correct them. The next chapter tells you about BASIC09's powerful debugging functions.

Chapter 5

The Debug Mode

The term *debug* refers to the process of finding programming errors and correcting them. BASIC09's debugging features include *symbolic* debugging capabilities that let you examine variable values and test and manipulate procedures.

With debug you can:

- Examine and change variables.
- Trace procedure execution. Debug lets you execute procedures and watch them run in slow motion.
- Pause procedure execution.
- Resume procedure execution.
- Set procedure *breakpoints* that automatically switch to the debug mode.
- Select and use degrees or radians for trigonometric functions.
- Perform calculations.
- Call NitrOS-9 system commands.

Entering the Debug Mode

You enter Debug:

- Automatically, whenever an error occurs during the execution of a procedure (unless you have included an ON ERROR GOTO statement to handle the error).
- Automatically, when a procedure executes a PAUSE statement.
- When you press `[CTRL][C]` during the execution of a procedure.

You can tell when BASIC09 enters the Debug mode by the appearance of the `D:` prompt. When you see `D:`, followed by the cursor, Debug is waiting for your command.

The following is a reference of all of the Debug commands and what they accomplish:

Command	Function
$	Calls NitrOS-9's command shell interpreter to run a program or an NitrOS-9 command. From the Debug prompt, type `$`, followed by the name of the program or command you want to execute. Example: `$list procedure_one[ENTER]`
BREAK	Sets a breakpoint immediately before the specified procedure. Use this command to re-enter Debug when one procedure calls another. If you have three procedures that call each other—Proc1, Proc2, and Proc3—and Proc3 doesn't seem to pass the correct values to Proc2 when it returns, set a breakpoint at Proc2. This causes BASIC09 to enter Debug before re-entering Proc2. You can then check your variable values. You use one breakpoint for each active procedure. Debug removes breakpoints immediately after encountering them. A procedure must run before you can set a breakpoint in it. Use BREAK to stop execution when a called procedure returns to a procedure previously executed. Example: `break proc2[ENTER]`
CONT	Causes procedure execution to continue. Example: `cont[ENTER]`
DEG/RAD	Selects either degrees or radians as the unit of measurement for trigonometric functions. DEG and RAD affect only the current procedure. Examples: `deg[ENTER]` `rad[ENTER]`

Command	Function
DIR	Displays the name, size, and variable storage requirements for each procedure in the workspace. The current working procedure has an asterisk before its name. All packed procedures have a dash before their names. DIR also shows the available memory in the workspace. If you provide a pathlist, DIR sends its data to the specified file. Examples: `dir[ENTER]` `dir procedures[ENTER]`
Q	Terminates execution of the procedure, closes any open paths, and exits to the System mode. Example: `q[ENTER]`
LET	Assigns a new value to a variable. You must specify variable names that are already initialized by your program. In the Debug mode, You cannot use LET to copy one array to another array as you can in BASIC procedures. Examples: `let a := 0[ENTER]` `let fruit := "oranges"[ENTER]`
LIST	Displays a source listing of the suspended procedure. The display is formatted and contains I-code addresses. An asterisk appears to the left of the last executed statement. Example: `list[ENTER]`
PRINT	Displays the values of variables used in the suspended procedure. You cannot introduce new variable names in the Debug mode, and you cannot display array or complex structures (although you can display individual array elements). The question mark short form of PRINT does not work; the whole word must be typed.

Command	Function
	You can print multiple variables at once in a single PRINT by separating them by commas (spaces them apart by 16 characters) or semi-colons (runs them together with no spaces). Example: `print fruit[ENTER]`
STATE	Lists the *nesting* order of active procedures. STATE displays the highest-level procedure at the bottom of the calling list. The lowest-level procedure is the suspended procedure. Example: `state[ENTER]`
STEP	Causes execution of the suspended procedure in specified increments. For example, typing `step 5[ENTER]` execute the equivalent of the next five statements. If you enter STEP without a increment value, the step rate is 1. Using step with the trace function lets you observe the source lines as they execute. Because compiled I-Code contains actual statement memory addresses, the *top* or *bottom* statements of loop structures execute only once. For example, in FOR/NEXT loops, FOR executes once, and the statement following FOR appears to be the top of the loop.
TRON/TROFF	Turns on or turns off the trace function. Trace on (TRON) causes the system to reconstruct the compiled code of each statement line into source code. Debug displays the source code before the statement is executed. If the statement causes the evaluation of one or more expressions, Debug displays each result following the statement. The result is preceded by an equal sign. The trace function is local to the current procedure. If the suspended procedure calls another procedure, Debug suspends the trace function until control returns to the original procedure. However, once you turn on trace for a procedure, it continues in effect until you turn it off. This means that if you turn trace on in a called procedure, and

another procedure subsequently calls it, trace continues to display the called procedure's operations.

Examples: `tron[ENTER]`
`troff[ENTER]`

When Things Go Wrong

Programming errors show up in two ways. Either your procedure produces incorrect results, or it terminates prematurely.

In the first instance, you can stop your procedure and enter Debug by pressing `[CTRL][C]`.

However, sometimes your program executes too quickly to allow you to stop it at the appropriate place. In this case, you can use the Edit mode to insert a PAUSE command where you wish the procedure to stop. PAUSE causes the procedure to halt execution and enter the Debug mode.

Once in Debug, you can use the PRINT command to examine the procedure variables. You can use LET to manipulate the variable values to determine where the error or errors occurred. Perhaps you forgot to initialize a variable or forgot to increase a loop counter.

Using the Trace Function

Sometimes, errors are more difficult to discover. If so, the next step is to use the trace function. To do this, type:

```
tron[ENTER]
```

Now press `[ENTER]`. Each time you press `[ENTER]`, Debug executes one line of the procedure. You can see the original source statement, and if an expression is evaluated, Debug prints the result of the expression, preceded by an equal sign.

In this manner, you can step through the entire procedure, or any part of it, examining variable values as you go.

What About Loops?

The STEP command is helpful if you find yourself tracing the operation of a loop. Once you determine that the loop works correctly, you can avoid tedious, step-by-step repetitions by turning trace off in using STEP to quickly run through the loop. Then, turn trace back on and resume single-step debugging. For instance, type:

```
troff[ENTER]
```

```
step 200[ENTER]
tron[ENTER]
```

In Multiple Procedures

Although the trace function is local to a procedure, you can pause and turn on the trace function in as many procedures as you wish. Trace continues to operate in each procedure until you turn it off using TROFF.

To cause a procedure to halt execution when it is called by another procedure, use the BREAK command.

Chapter 6

Data and Variables

Data Types

Data is information on which a computer performs its operations. Data is always numeric but, depending on your computer application, it can represent values, symbols, or alphabetic characters. This means that the same items of *physical* data can have very different *logical* meanings, depending on how a program interprets it.

For instance, 65 can represent:

- A numeric value to be used in a calculation.
- The location of a memory address.
- The *offset* of a memory location.
- The two character symbols 6 and 5.
- The character A in the ASCII table. ASCII is the abbreviation for the American Standard Code for Information Interchange.

Because of the differences in how BASIC09 uses data, the system lets you define five types of data. For instance, there are three ways to represent numbers. Each has its own advantages and disadvantages. The decision to use one way or another depends on the specific program you are developing. The five BASIC09 data types are byte, integer, real, string, and boolean.

In addition to the preceding data types, there are *complex data types* (also known as *record* types) that you can define. The manual discusses complex data structures at the end of this chapter.

The *byte*, *integer*, and *real* data types represent numbers.

The *string* data type represents character data (alphabet, punctuation, numeric characters, and other symbols). The default length of strings is 32 characters. Using the DIM statement, you can specify strings varying in length from 0, a *null* string, to 32,767 characters (or until you reach the capacity of memory available to BASIC09, whichever occurs first).

The *boolean* data type represents the logical value, TRUE or FALSE.

You can create arrays (lists) of these data types with one, two, or three dimensions. The following table shows the data types and their characteristics:

Type	Allowable Values	Memory Requirements
BYTE	Whole numbers (0 to 255)	One byte
INTEGER	Whole numbers (-32768 to 32767)	Two bytes
REAL	Floating point ($\pm 1 * 10^{\pm 38}$)	Five bytes
STRING	Letters, digits, punctuation	One byte per character
BOOLEAN	True or false	One byte

Real numbers appear to be the most versatile. They have the greatest range and are floating point. However, arithmetic operations involving real numbers execute much more slowly than those involving integer or byte values. Real numbers also take up considerably more memory storage space than the other two numeric data types.

Arithmetic involving byte values is not appreciably faster than arithmetic involving integers, but byte data conserves memory.

If you do not specify the type of variable (a symbolic name for a value) in a DIM statement, BASIC09 assumes the variable is real.

The Byte Data Type

Byte variables hold unsigned eight-bit data (integers in the range of 0 through 255). Using byte values in computations, BASIC09 converts the byte values to 16-bit integer values. If you store an integer value that is too large for the byte range, BASIC09 stores only the least-significant eight bits (a value of 255 or less), and does not return an error.

The Integer Data Type

Integer variables require two bytes (16 bits) of storage. They can fall in the range -32768 to 32767. If a calculation involves both integer values and real values, BASIC09 presents the result of the calculation as a real number.

You can use hexadecimal values in integer data. To do so, precede the value with the dollar sign ($). For instance, to represent the decimal value 199 as hexadecimal, type `$C7`. The hexadecimal value range is `$0000` through `$FFFF`.

If you give an integer variable a value that is outside the integer range (greater than 32767 or less than -32768), BASIC09 does not produce an error. Instead, it *wraps around* the value range. For instance, the calculation 32767 +1 produces a result of -32768.

This means that numeric comparisons made on values in the range 32768 through 65535 deal with negative numbers. You should limit such comparisons to tests for equality or non-equality. Functions such as LAND, LNOT, LOR and LXOR use integer values but produce results on a non-numeric, bit-by-bit, basis.

Division of an integer by another integer results in an integer. BASIC09 discards any remainder.

The Real Data Type

If you do not assign a data type to a variable, BASIC09 assumes the variable is real. However, programs are easier to understand if you define all variable types.

BASIC09 stores as real values any constants that have decimal points. If a constant does not have a decimal point, BASIC09 stores it as an integer, except in the case where the constant value exceeds the range of an integer. Then, it is converted to a real value and is displayed with a decimal point.

BASIC09 requires five consecutive bytes to store real numbers. The first byte is the exponent, in binary two's complement. The next four bytes are the binary sign and magnitude of the mantissa. The mantissa is in the first 31 bits; the sign of the mantissa is in the last (least significant) bit of the last byte. The following illustration shows the memory storage of a real number:

Internal Representation of Real Numbers

The exponent covers the range $2.938735877 \times 10^{-39}$ (2^{-128}) through $1.701411835 \times 10^{38}$ (2^{127}) as powers of 2. Operations that result in values out of the representation range cause an overflow or underflow error. You can handle such errors using the ON ERROR command.

The mantissa covers the range 0.5 through .9999999995 In steps of 2^{-31}. This means that real numbers can represent values .0000000005 apart. BASIC09 rounds operation values that fall between these points to the nearest point.

Because floating point arithmetic is inherently inexact, a sequence of operations can produce a cumulative error. Proper rounding, as implemented in BASIC09, reduces the effect of this problem, but cannot eliminate it. When using real quantities in comparisons, be sure your computations can produce the exact value you desire.

The String Data Type

A string is a variable-length sequence of ASCII values. The length can vary from 0, a *null* string, to 32,767 characters (or until you reach the capacity of memory available to BASIC09, whichever occurs first).

You can define a string variable either explicitly, using the DIM statement, or implicitly by appending the dollar sign (`$`) to the variable identifier (variable name). For example, title$ implicitly identifies a string variable.

Unless you specify otherwise, BASIC09 assigns a maximum string length of 32 characters. Using the DIM statement, you can specify a maximum length either less than or greater than 32. To conserve memory, use DIM to assign only the maximum length you need for any string variable.

The beginning of a string is always character 1. The BASE statement, which sets numeric variable base numbers as either 0 or 1, does not affect string variables.

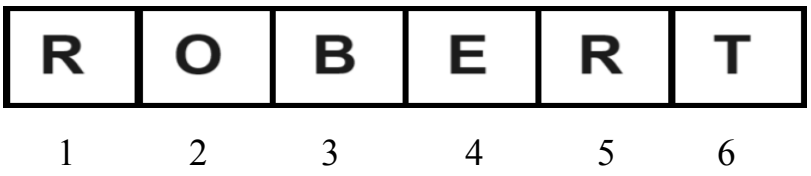
If an operation results in a string too long to fit in the assigned maximum storage space, the system truncates the string on the right. It does not produce an error.

Storing storage is fixed at the dimensioned length. The sequence of actual string byte values is terminated by the value of $FF (255), or by the maximum length allotted to the string. Any unused storage after the termination byte allows the stored string to expand and contract within its assigned length.

The following example shows the internal storage of a variable dimensioned as `string[6]` and assigned the value "SAM". Note that byte 4 contains the string terminator $FF. The string does not use bytes following $FF.

byte: 1 2 3 4 5 6

If you assign the value "ROBERT" to the variable, BASIC09 does not need to terminate the string with $FF because the string is full:



The way BASIC09 handles string storage is important when you write programs. If you do not specify a length for strings you define, the system uses the default length 32. As you can see, this wastes computer memory if you store strings of only four or five characters.

The Boolean Data Type

A boolean operation always returns the character string "TRUE" or "FALSE". You cannot use the boolean data type for numeric computation—only for comparison logic. NOTE: TRUE is stored internally as a byte value of 255 and FALSE as 0. This can come in handy if you call a procedure where you receive data as BOOLEAN (even if sent as BYTE or STRING), or if you use PEEK(ADDR(boolean variable)). If BYTE values are received by a BASIC09 procedure as BOOLEAN, BASIC09 will consider a byte value of 0 as FALSE and all others as TRUE.

Do not confuse the boolean operations AND, OR, XOR, and NOT (which operate on the boolean values TRUE and FALSE) with the logical functions LAND, LOR, LXOR and LNOT (which use integer values to produce numeric results on a bit-by-bit basis). An attempt to store a non-boolean value In a boolean variable, causes an error.

Automatic Type Conversion

When an operation mixes numeric data types (byte, integer, or real values), BASIC09 automatically and temporarily converts the values to the type necessary to retain accuracy. This conversion lets you use numeric quantities of mixed types in most calculations.

The system returns a type-mismatch error when an expression includes types that cannot legally mix. These errors are reported by the second compiler pass, which occurs automatically when you exit the edit mode.

Because type conversion takes additional execution time, you can speed calculations by using values of a single type.

Constants

Constants are values in a program that do not change. They can use any of the five data types. The following are examples of constants in a procedure:

```
HOME$="Fort Worth"
VALUE$="25,000"
VALUE=25
PAYMENT=99.99
ANSWER="TRUE"
MEMORY=$0CFF
PRINT "The End"
```

Numeric constants are either integers or real numbers. If a numeric constant includes a decimal point or uses the "E format" exponential form, it causes BASIC09 to store the number in the real format, even if it could store the number in integer or byte format.

You can use this feature to *force* a real format. For instance, to make the number 12 a real number, type it as 12.0. You might want to force real values in this way when all other values in an expression are real so that BASIC09 does not have to do a time-consuming type conversion at run time.

BASIC09 also stores as real numbers any numbers that do not have decimal points but that are too large to store as integers. Here are some examples of legal real constants:

```
1.0             9.8433218      -.01
-999.000099     100000000      5644.34532
1.95E+12        -99999.9E-33
```

BASIC09 treats numbers that do not have a decimal point and are in the range -32768 through +32767 as integers. You must always precede hexadecimal numbers with a dollar sign.

Following are examples of legal integer constants:

```
12              -3000          55
$20             $FF            $09
0               -12            -32768
```

String Constants

A string constant consists of a sequence of characters enclosed in double quotation marks, such as:

```
"The End"
```

To place a string constant into a string type variable, use the = symbol in this manner:

```
TITLE$ = "Masters Of Magic"
```

To include double quotation marks within a string, use two sets of double quotation marks, like this:

```
"An ""older man"" is wiser."
```

A string can contain characters that have ASCII values in the range 0 through 255.

Variables

In BASIC09, a variable is *local* to the procedure in which it is defined. A variable defined in one procedure has no meaning in another procedure unless you use the RUN and PARAM statements to pass the variable when you call the other procedure.

The local nature of variables lets you use the same variable name in more than one procedure and, unless you specify otherwise, have the variables operate independently of each other.

You can assign variables using either the LET statement with the assignment operator (=), or by using the assignment operator alone. Alternately, you can also use the := assignment operator. For instance, both of the following command lines are legal:

```
LET PAYMENT=44.50
PAYMENT:=44.50
```

When you call a procedure, BASIC09 allocates storage for the procedure's variables. It is not possible to force a variable to occupy an absolute address in memory. When you exit a procedure, the system returns the storage allocated for the variables, and you lose the stored values.

If you write a procedure to call itself (a *recursive* procedure), the call creates separate storage space for variables.

> **Note:** Unlike other BASICs, BASIC09 does not automatically initialized variables by setting them to zero. When you execute a procedure, all variables, arrays, and structures have random values. Your procedure must initialize the variables you specify to the values you require.

Passing Variables

When one procedure passes variable values to another procedure, BASIC09 refers to the passed variables as *parameters*. You can pass variables either by *reference* or by *value*.

Passing By Reference

BASIC09 **does not** protect variables passed by reference. Therefore, the called procedure can change the values and return the new values.

To pass parameters by reference:

```
RUN ADDCOLUMN(x)
```
pass by reference

The system evaluates the storage address of each passed variable, and sends the variable to the called procedure. The called procedure associates the storage addresses with the names in its local PARAM statement. It then uses the storage area as though it had created it locally. This means that it can change the value of the parameter before returning it to the calling procedure.

Passing By Value

BASIC09 **does** protect variables passed by value, so, the called procedure cannot change them.

To pass parameters by value, write the value to be passed as an expression. BASIC09 evaluates the expression at the time of the call. To use a variable in an expression without changing its value, use null constants, such as 0 for a number or "" for a string, in this manner:

```
RUN ADDCOLUMN(x+0)
```
passes the value of x by value

```
RUN TRANSLATE(w$+"")
```
passes the contents of w$ by value

To pass parameters by value, BASIC09 creates a temporary variable. It places the result of the expression in the temporary variable and sends the address to the called procedure. This means that the value given to the called procedure is a *copy* of the result of the expression, and the called procedure cannot change the original value.

The results of expressions containing numeric constants are either integer or real values; there are no byte constants. To send byte-type variables to a procedure, pass the values by reference. Therefore, if a RUN statement evaluates an integer as a perimeter and sends it to a byte-type variable, the byte Variable uses only the high-order byte of the two-byte integer.

Arrays

An *array* is a group of related data values stored consecutively in memory. The system knows the entire group by a variable name. Each data value is an *element*. You use a *subscript* to refer to any element of the array. For example, an array named Graf might contain five elements referred to as:

```
Graf(1)     Graf(2)  Graf(3)  Graf(4)  Graf(5)
```

You can use each of these elements to store a different value, such as:

```
Graf(1) = 25
Graf(2) = 47
Graf(3) = 39
Graf(4) = 18
Graf(5) = 50
```

Note: Normally, array elements start with 1 in BASIC09. However, you can use the BASE statement to cause array elements to begin at 0.

The previous example illustrates a single-dimensioned array. The elements are arranged in one row and only one subscript is used for each element.

The following procedure lets you type values for a GRAF array and displays the results in a simple graph.

```
PROCEDURE GRAF
□DIM GRAF(5):REAL
□PRINT CHR$(12)
□FOR T=1 TO 5
□PRINT "Value for Item #"; T; "□";
□INPUT GRAF(T)
□NEXT T
□PRINT
□PRINT
□PRINT "This is how your graph stacks up..."
□PRINT
□FOR T=1 TO 5
□PRINT "Item #"; T; "□";
□FOR U=1 TO GRAF(T)
□PRINT CHR$(79);
□NEXT U
□PRINT
```

```
 NEXT T
 PRINT
 END
```

This procedure uses a single dimension array—in effect, a list.

You can also create arrays with more than one dimension — more than one element for each row. You might use a two-dimensioned array in a procedure to store names and addresses. Instead of creating separate arrays for the name, address, and zip code, you could set up one array with two dimensions.

The following procedure, used to enter the names of a company's employees, shows how this might be done. See the second line for the DIM syntax. When you run the procedure, it asks you for a name, address, and zip code for each of 10 employees. After you type the information for all the entries, the procedure displays the information on the screen.

```
PROCEDURE Names
 DIM NAME(10,3):STRING
 PRINT CHR$(12)
 BASE 0
 FOR T=0 TO 9
 PRINT "Type Employee Name No."; T; ": ";
 INPUT NAME(T,0)
 PRINT "Type Employee Address No."; T; ": ";
 INPUT NAME(T,1)
 PRINT "Type Employee Zip Code No."; T; ": ";
 INPUT NAME(T,2)
 NEXT T
 PRINT CHR$(12)
 PRINT "And the names are..."
 PRINT
 FOR T=0 TO 9
 PRINT NAME(T,0); " "; NAME(T,1); " "; NAME(T,2)
 NEXT T
 END
```

The DIM statement reserves space in memory for a string array named NAME with two dimensions. As you enter data, the Name field is stored in NAME(0,0), NAME(1,0), NAME(2,0), and so on. The Address field is stored in NAME(0,1), NAME(1,1), NAME(2,1), and so on. The Zip Code field is stored in NAME(0,2),

NAME(1,2), NAME(2,2), and so on. This continues until you fill the *grid*, 10 entries with three items each.

You can also create the arrays with three dimensions. The following procedure adds one more dimension that keeps track of each employee's company. It dimensions Name$ as Name$(2,10,3). The first dimension contains either 0 or 1 to indicate which company the employee works for.

```
PROCEDURE Names2
DIM NAME$(2,10,3):STRING
PRINT CHR$(12)
BASE 0
FOR X=0 TO 1
PRINT
PRINT
FOR T=0 TO 9
PRINT
IF X=0 THEN
PRINT "Type a Wiggleworth Company Employee Name"
ELSE
PRINT "Type a Putforth Company Employee Name"
ENDIF
PRINT "Type Name No."; T; ": ";
INPUT NAME$(X,T,0)
PRINT "Type Address No."; T; ": ";
INPUT NAME$(X,T,1)
PRINT "Type Zip Code No."; T; ": ";
INPUT NAME$(X,T,2)
NEXT T
NEXT X
PRINT CHR$(12)
PRINT "The Wiggleworth employees are..."
PRINT
X=0
FOR T=0 TO 9
PRINT NAME$(X,T,0); "□"; NAME$(X,T,1); "□";
NAME$(X,T,2)
NEXT T
PRINT
PRINT "The Putforth employees are..."
```

```
PRINT
X=1
FOR T=0 TO 9
PRINT NAME$(X,T,0); " "; NAME$(X,T,1); " ";
NAME$(X,T,2)
NEXT T
END
```

The easiest way to understand three dimensional arrays is to consider the first dimension as a *page*. In other words if the first dimension in the string is 0, the employee is on the Wiggleworth Company's page. If the first dimension in the string is 1, the employee is on the Putforth Company's page.

Complex Data Types

In addition to the five standard data types, you can create your own data types. Using the TYPE statement, you can define a new data type as a *vector* (a single-dimensioned array) of any previously defined type.

For example, in the previous procedure, the NAME variable can only contain one data type, the string type. However, using the TYPE statement you can create a variable that accepts several data types. Suppose you create an employee list procedure that uses the following variables, of the following size and types:

Name	Length	Contents	Type
Name	25	employee name	string
Street	20	street address	string
City	10	city of address	string
Zip Code	—	address zip code	integer
Sex	—	false = male, true = female	boolean
Age	—	employee age	byte

You can combine all these variables into one complex data type. To do so, dimension the variables within a TYPE statement like this:

```
TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
```

This creates a new BASIC09 type, called EMPLOYEE. EMPLOYEE requires its variables to have six fields of the name, size, and type shown in the previous chart.

Once you create the new data type, you can define variables to use it. For instance, the following source instruction defines WORKER as type EMPLOYEE with 10 elements in the array:

```
DIM WORKER(10):EMPLOYEE
```

To put the EMPLOYEE data type to work, collect your data with INPUT commands. Then, store the data into the new WORKER array. The following procedure demonstrates how you might do this:

```
PROCEDURE worker
REM ▯▯▯▯▯▯▯▯▯Dimension variables for input
DIM NM:STRING[25]
DIM ST:STRING[20]
DIM CTY:STRING[10]
DIM ZP:REAL
DIM SX:BOOLEAN
DIM AG:BYTE
REM Create new type and array using new type
TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
DIM WORKER(10):EMPLOYEE
REM
FOR T=1 TO 10
INPUT "Name:▯",NM
INPUT "Street:▯",ST
INPUT "City:▯",CTY
INPUT "Zip:▯",ZP
INPUT "Sex:▯",SX
INPUT "Age:▯",AG
REM ▯▯▯▯▯▯▯Store input ln the Worker array
using field names
WORKER(T).NAME=NM
WORKER(T).STREET=ST
WORKER(T).CITY=CTY
WORKER(T).ZIP=ZP
WORKER(T).SEX=SX
WORKER(T).AGE=AG
PRINT
PRINT "* * * * * * * * * * * * * * * * * * * *"
PRINT
NEXT T
PRINT CHR$(12);
PRINT "The names in your files now are..."
```

```
PRINT
FOR T=1 TO 10
PRINT WORKER(T).NAME
PRINT WORKER(T).STREET
PRINT WORKER(T).CITY
PRINT WORKER(T).ZIP
IF WORKER(T).SEX=TRUE THEN
PRINT "Female"
ELSE
PRINT "Male"
ENDIF
PRINT WORKER(T).AGE
PRINT
PRINT "* * * * * * * * * * * * * * * * * * * *"
PRINT
NEXT T
```

Note that the SEX field is defined as BOOLEAN. This means that you can respond only in two ways, TRUE or FALSE. The method of input requires only one byte of storage. To use this data you need to handle it So TRUE and FALSE indicate male and female.

Complex data types can contain more than one field. Each field can be of any data type. You referenced the fields of a complex data type by typing the variable name, its array index, a period (.), and the field name. The following lines, from the worker procedure, shows how BASIC09 stores the data from the input lines into the WORKER variable:

```
WORKER(T).HAME=NM
WORKER(T).STREET=ST
WORKER(T).CITY=CTY
WORKER(T).ZIP=ZP
WORKER(T).SEX=SX
WORKER(T).AGE=AG
```

These lines store the values in the variables NM, ST, , CTY, ZP, SX and AG into the NAME, STREET, , CITY, ZIP, SEX and AGE Fields of the WORKER variable. This operation is within a FOR/NEXT loop that uses T as a counter. In the procedure T can refer to a value in the range 1 to 10.

The procedure uses the same type of operation to extract the data from the complex data type variable:

```
PRINT WORKER(T).NAME
PRINT WORKER(T).STREET
PRINT WORKER(T).CITY
PRINT WORKER(T).ZIP
IF WORKER(T).SEX=TRUE THEN
PRINT "Female"
ELSE
PRINT "Male"
ENDIF
PRINT WORKER(T).AGE
```

Using the same methods, you can create complex data types that combine other complex data types and standard data types.

The elements of a complex structure can be copied to another similar structure. Using a single assignment operator, you can write an entire structure to, or read an entire structure from, mass storage as a single entity. For example:

```
PUT #2, WORKER(T)
```

Because the system defines the elements of complex-type storage during compilation, it need not do so during *runtime*. This means that BASIC09 can reference complex structures faster than it can reference arrays.

Chapter 7

Expressions, Operators, and Functions

Manipulating Data

BASIC09 uses *expressions* to manipulate data. (Expressions are pieces of data connected by operators.)

An *operator* is a symbol or a word that signifies some action to be performed on the specified data. Each data item is a value.

Expressions

When an expression is evaluated, the result is a value of some data type (real, integer, string, byte, or boolean).

An expression might look like this:

First Value	First Operator	Second Value	Second Operator	Result
6	+	5	=	11

or like this:

First Value	First Operator	Second Value	Second Operator	Result
“Seaside”	+	“Villa”	=	Seaside Villa

When BASIC09 evaluates an expression, it copies each value onto an expression stack. Functions and operators take their input values from this stack and return their results to it. Many expressions result in assignments, as do the examples shown. Then BASIC09 makes the resulting assignment only after it computes the entire expression. This lets you use the variable that is being modified as one of the values in the expression, such as in this example:

```
X=X+1
```

The result of an expression is always one of the five BASIC09 data types. However, you can often mix data types within an expression and, in some cases, the result of an expression is of a different data type than any of the values in the expression. Such is the case if you use the less-than symbol (<), in this manner:

```
24 < 188
```

The less-than operator compares two integer values. The result of the comparison is boolean; in this case, the value is TRUE.

Type Conversion

Because BASIC09 performs automatic type conversion of values, you can mix any of the three numeric data types in an expression. When you mix numeric data types, the result is always of the same type as the value having the largest representation, in this order: real < integer < byte.

You can use any numeric type in an expression that produces a real number. If you want an expression to produce a byte or integer type value, the result must be small enough to fit the desired type.

Operators

BASIC09 has operators to deal with all types of data. Each operator, except NOT and negation (unary -), takes two values or operands, and performs an operation to produce a result. NOT can accept only one value. The following table lists the operators available and the types of data they accept and produce.

Because the same operators function on the three types of numeric data (byte, integer, and real), these types are referred to by the operand type "numeric."

BASIC09 Expression Operators

Operator	Function	Operand Type	Result Type
-	Negation	numeric	numeric
~or **	Exponentiation	numeric	numeric
*	Multiplication	numeric	numeric
/	Division	numeric	numeric
+	Addition	numeric	numeric
-	Subtraction	numeric	numeric

NOT	Logical Negation	boolean	boolean
AND	Logical AND	boolean	boolean
OR	Logical OR	boolean	boolean
XOR	Logical Exclusive OR	boolean	boolean
+	Concatenation	string	string
=	Equal to	all types	boolean
<> or ><	Not equal to	all types	boolean
<	Less than	numeric, string†	boolean
<= or =<	Less than or equal	numeric, string†	boolean
>	Greater than	numeric, string†	boolean
>= or =>	Greater than or equal	numeric, string†	boolean

† When comparing strings, BASIC09 uses the ASCII values of characters as the basis for comparison. Therefore, 0 < 1, 9 < A, A < B, A < b, b < z and so on.

Arithmetic Operators

Arithmetic operators perform operations on numeric data. Therefore, both operands in the expression must be numeric. The following table lists the arithmetic operators.

Negation The single dash negates a number's sign:
-10 is *negative* 10.

Exponentiation Use a caret (^) or two asterisks (**) to raise a number to a power:
2^3 is 8 (2x 2 x 2).
Similarly, 2**3 is 8.

Multiplication A single asterisk causes multiplication:
2*3 is 6.

Division A slash causes division: 6 / 2 is 3.

Addition The plus sign causes addition:
3 + 3 is 6.

Subtraction A dash causes subtraction: 6 - 3 is 2.

Hierarchy of Operators

BASIC09 uses the standard hierarchy of operations when calculating expressions with multiple operators. This means that BASIC09 has an order in which it performs calculations involving more than one operator.

The following BASIC09 operators are listed in order of precedence:

```
NOT - (negate}
^ **
* /
+ -
> < <> = >= <=
AND
OR XOR
```

Also, BASIC09:

- Performs operations enclosed in parentheses before operations not in parentheses.
- Performs the leftmost operations first when two or more operations are of equal precedence.

You can use parentheses to override this standard precedence.

For example:

```
2 + 1 * 3 = 5
```

but

```
(2 + 1) * 3 = 9
```

The following examples show BASIC09 expressions on the left, and the way BASIC09 evaluates them on the right. You can enter the expressions in either form, but the decompiler generates the simpler form, shown on the left.

BASIC09 Representation	Equivalent Form
a=b+c**2/d	a=b+((c**2)/d)
a=b>c AND d>e OR c=e	a=((b>c) AND (d>e)) OR (e=e)
a=(b+c+d)/e	a=((b+c)+d)/e
a=b**c**d/e	a=(b**(c**d))/e
a=-(b)**2	a=(-b)**2

Relational Operators

Relational operators make logical comparisons of any type of data and return a result of either TRUE or FALSE. An explanation of the relational operators follows. All relational operators have equal precedence.

=	Equal. Returns TRUE if both operands are equal, or FALSE if they are not equal.
<	Less than: Returns TRUE if the first operand is less than the second, or FALSE if is not.
>	Greater than: Returns TRUE if the first operand is greater than the second, or FALSE if it is not.
<> or ><	Unequal: Returns TRUE if the operands are not equal or FALSE if they are.
<= or =<	Less than or equal to: Returns TRUE if the first operand is less than or equal to the second operand. Otherwise, the operation returns FALSE.
>= or =>	Greater than or equal to: Returns TRUE if the first operand is greater than or equal to the second. Otherwise, the operation returns FALSE.

You normally use relational operators in IF/THEN statements. For example, if your procedure has two numeric variables, Payments and Income, you might include command lines like this:

```
IF PAYMENTS > INCOME THEN
  PRINT "You're Broke!"
ENDIF
```

When you combine arithmetic and relational operators in the same expression, BASIC09 evaluates the arithmetic operations first. For example:

```
IF X*Y/2 <= 14 THEN
  PRINT "Average Score is "; X*Y/2
ENDIF
```

BASIC09 performs the arithmetic operation x+y/2, then compares the result with the value 14.

When you use relational operators with strings, BASIC09 compares the strings character by character. When it finds two characters that do not match, it checks to see which character has the lower ASCII code value. The string containing the character with the lower value comes first.

Consider this example:

```
PRINT "hunt" > "hung"
```

BASIC09 compares each character in each string. Because the first three characters are the same, the result of the operation is based on the comparison of t and g. Because t (ASCII value = 116) is "greater than" g (ASCII value = 103), the command prints TRUE.

String Operators

The string operator is the plus sign (+). This symbol appends one string to another. All operands must be strings, and the resulting value is one string. Examine, for example, the following line, which appends three strings:

```
PRINT "My friends are " + "Jack and " + "Jill."
```

It prints: `My friends are Jack and Jill.`

Logical Operators

The logical, or boolean, operators make logical comparisons of boolean values. The following table describes the results yielded by each logical operator given the specified TRUE/FALSE values:

Operator	Meaning of Operation	First Operand	Second Operand	Result
NOT	The result is the opposite of the operand.	TRUE		FALSE
		FALSE		TRUE
AND	When both values are TRUE, the result is TRUE. Otherwise, the result is FALSE.	TRUE	TRUE	TRUE
		TRUE	FALSE	FALSE
		FALSE	TRUE	FALSE
		FALSE	FALSE	FALSE
OR	When both values are FALSE, the result is FALSE. Otherwise, the result is TRUE.	FALSE	FALSE	FALSE
		TRUE	TRUE	TRUE
		TRUE	FALSE	TRUE
		FALSE	TRUE	TRUE
XOR	When only one of the values is TRUE, the result is TRUE. Otherwise the result is FALSE.	TRUE	FALSE	TRUE
		FALSE	TRUE	TRUE
		TRUE	TRUE	FALSE
		FALSE	FALSE	FALSE

Use logical operators in IF/THEN statements such as:

```
IF PAYMENTS < INCOME AND INCOME+SAVINGS >
PAYMENTS THEN
```

```
    PRINT "You'll have to use your savings to get
out of this mess."
ENDIF
```

Functions

Functions are operation sequences the system performs on data. In a statement, BASIC09 performs functions first. Chapter 11, "Command Reference," describes the following functions.

Functions returning results of type real:	
SIN	Calculates the trigonometric sine of a number.
COS	Calculates the trigonometric cosine of a number.
TAN	Calculates the trigonometric tangent of a number.
ASN	Calculates the trigonometric arcsine of a number.
ACS	Calculates the trigonometric arccosine of a number.
ATN	Calculates the trigonometric arctangent of a number.
LOG	Calculates the natural logarithm (base e) of a number.
LOG10	Calculates the logarithm (base 10) of a number.
EXP	Calculates e (2.71828183) raised to the specified positive power.
FLOAT	Converts byte or integer type numbers to real numbers.
INT	Calculates the largest whole number less than or equal to the specified number.
PI	Represents the constant 3.14159265.
SQR	Calculates the square root of a positive number.
SQRT	Calculates the square root of a positive number. Its function is identical to SQR.
RND	Returns a random number.

Functions returning results of any numeric type: The resulting type depends on the input type.	
ABS	Calculates the absolute value of a number.
SGN	Returns a value to indicate the sign of the specified number (-1 if the number is less than 0, 0 if the number is 0, or 1 if the number is greater than 0).
SQ	Calculates the square of a number.
VAL	Converts a string to a numeric value.

Functions returning results of type integer or type byte:	
FIX	Rounds a real number and converts it to an integer.
MOD	Calculates the modulus (remainder) of two numbers.
ADDR	Returns the absolute memory address of a variable, an array, or a

	structure.
SIZE	Returns (in bytes) the storage size of a variable, an array, or a structure.
ERR	Returns the error code of the most recent error.
PEEK	Returns the byte value at a specified memory address.
POS	Returns the current character position of the print buffer.
ASC	Returns the numeric value (ASCII code) of a string character.
LEN	Returns the length of a string.
SUBSTR	Returns the starting position of the specified substring within a string, or returns 0 if it cannot find the substring.
Functions performing bit-by-bit logical operations on integer or byte data and returning integer results. Do not confuse these functions with boolean type operators.	
LAND	Calculates the logical AND of two values.
LOR	Calculates the logical OR of two values.
LXOR	Calculates the logical EXCLUSIVE OR of two values.
LNOT	Calculates the logical NOT of a value.
Functions returning a result of type string:	
CHR$	Returns the character having a specified ASCII value.
DATE$	Returns the system's current date and time.
LEFT$	Returns the specified number of characters beginning at the leftmost character of the specified string.
RIGHT$	Returns the specified number of characters beginning at the rightmost character of the specified string and counting backward.
MID$	Returns the specified number of characters starting at the specified position in a string.
STR$	Converts numeric type data to string type.
TRIM$	Removes trailing spaces from the specified string.
Functions returning results of type boolean:	
TRUE	Always returns TRUE.
FALSE	Always returns FALSE.
EOF	Tests for the end of a disk file. Returns TRUE when the end of the file occurs.

Chapter 8

Disk Files

When you tell NitrOS-9 or BASIC09 to store (save) data on a disk, it stores the data in a *logical* block called a *file*. The term logical means that, although the system might store portions of a file's data in several different disk locations, it keeps track of every location and treats the scattered data as though it occupied a single block. It does this automatically and you never need to worry about how the data is stored. File data can be binary data, textual data (ASCII characters), or any other useful information.

Because NitrOS-9 handles all hardware input/output devices (disk drives, printers, terminals, and so on) in the same manner, you can send data to any of these devices in the same way. This means you can send the same information to several devices by changing the path the data follows. For example, you can test a procedure that communicates with a terminal by transferring data to and from a disk drive.

BASIC09 normally works with two types of files—sequential files and random access files. The following chart shows file-access options, their purposes, and the keywords with which to use them:

Types of Access for Files

Access Type	Function	Use with
DIR	Opens a directory file for reading. Use only with READ.	OPEN
EXEC	Specifies that the file to open or create is in the execution directory, rather than the data directory.	OPEN CREATE
READ	Lets you read data from the specified file or device.	OPEN CREATE
WRITE	Lets you write data to the specified file or device.	OPEN CREATE

UPDATE	Lets you read data from and write data to the specified file or device.	OPEN CREATE

Sequential Files

Sequential files send or receive (WRITE or READ) textual data in order, the second item following the first, and so on. You can access sequential data only in the same order as you originally stored it. To read from or write to a particular section of a file, you must first read through all the preceding data in the file, starting from the beginning.

BASIC09 stores sequential file data as ASCII characters. Each block of data is separated by a *delimiter* consisting of a carriage-return character (ASCII character 13). Because BASIC09 uses this delimiter to determine the end of a *record*, sequential files can contain records of varying length.

Use the WRITE and READ commands to store and retrieve data in sequential files. A WRITE command causes BASIC09 to transfer specified data to a specified file, ending the data with a carriage return. A READ command causes BASIC09 to load from the specified file the next block of data, stopping when it reaches a carriage return.

Sequential File Creation, Storage, and Retrieval

BASIC09 uses the CREATE command to establish both sequential and random access files. A CREATE statement contains:

- The keyword CREATE.
- A path number variable in which BASIC09 stores the number of the path it opens to the new file.
- A comma, followed by the name of the file to create.
- An optional colon, followed by the access mode. If you do not specify an access mode, BASIC09 automatically opens the created file in the UPDATE mode.

The following procedure shows how to create a file and write data into it:

```
PROCEDURE makefile
DIM PATH:BYTE \                        (* establishes a variable
REM                                    for the path number to the file
CREATE #PATH,"TEST":WRITE \            (* creates the file TEST
WRITE #PATH,"This is a test" \         (* writes data to the file
WRITE #PATH,"of sequential files." \(* writes another line of data
CLOSE #PATH \                          (* closes the path to the file
SHELL "LIST TEST" \                    (* displays the file contents
END
```

The first line of the procedure dimensions a variable (Path) to hold the number of the path that CREATE opens. This variable should be of byte or integer type.

When you establish a new file with CREATE, you automatically open a path to the file. You do not need to use the OPEN command.

The preceding procedure writes two lines into a file named Test. It then closes the path and uses the NitrOS-9 LIST command to display the contents of the newly created file. You see that the data is successfully stored on disk.

The next procedure shows how to reopen an existing file for sequential access, read the contents of the file, and append data to the end of the file.

The only way to move the file pointer to the end of a sequential file is to read all the data already in the file. Once the pointer is at the end of the file, you can add data.

```
PROCEDURE append
DIM PATH:BYTE \              (* dimension variable to hold the
REM                           number of the path to the opened file.
OPEN #PATH, "TEST":UPDATE \(* open file for reading and writing.
READ #PATH, line$ \          (* read the first element of the file.
READ #PATH, 1ine$ \          (* read the next (the last) element.
WRITE #PATH,"This is a test" \(* write one new line to the file.
WRITE  #PATH,"of  appending  to  a  sequential  file."  \(*  write
another.
CLOSE #PATH \                (* close the path.
SHELL "LIST TEST" \          (* display the file with the new lines.
END
```

Because the Test file already exists, this procedure uses OPEN to establish a path to the file. It uses the UPDATE mode of file access because it needs to both read from and write to the file.

The two READ statements read the file's contents and, as a result, move the file pointer to the end of the file. The WRITE statements then append two new lines. After closing the path, the procedure calls on the NitrOS-9 LIST command to display the contents of the file, with its appended lines.

Changing Data in a Sequential File

You can also change data anywhere in a sequential file. However, if your changes are longer than the original data, the operation destroys part of the file. To change data in a sequential file, read the data preceding what you want to change, and write the new data to the file in this manner:

```
PROCEDURE replace
DIM PATH:BYTE
OPEN #PATH, "TEST":UPDATE
READ #PATH, Line$
READ #PATH, Line$
WRITE #PATH,"Let's put new" \     (* write over existing 3rd and
WRITE #PATH,"words into the old sequential file." \(* 4th lines.
CLOSE #PATH
SHELL "LIST TEST"
END
```

Notice that the total amount of data in the two new lines is exactly the same as in the two old lines. You can replace an existing line with fewer characters by padding the new data with spaces. However, if you try to replace existing lines with longer lines, the new lines write over and destroy other data in the file.

The above example procedures use the SHELL statement to execute a NitrOS-9 command (LIST). Using the SHELL statement is not the best way to perform tasks within a BASIC09 procedure. Below is a better alternative for use with the above examples.

Replace the `SHELL "LIST TEST"` statements in the above procedures with:

```
RUN ListFile("TEST")
```

Then add a new procedure to the workspace:

```
PROCEDURE ListFile
DIM PATH:BYTE
PARAM FILENAME$
OPEN #PATH,FILENAME$:READ
REPEAT
READ #PATH,LINE$
PRINT LINE$
UNTIL EOF(#PATH)
CLOSE #PATH
END
```

You will learn more about loops in Chapter 11, BASIC09 Command Reference.

INPUT and Sequential Files

Although you can also use the INPUT command with sequential files, doing so might put unwanted data into them. When a procedure encounters INPUT, it suspends execution and sends a question mark (?) to the screen. This feature makes INPUT both an input and output statement. Therefore, if you open a file using the UPDATE mode, INPUT writes its prompts to the file, destroying data. If you specify text to be displayed with the INPUT command, INPUT writes this text to the file also.

Random Access Files

Random access files store data in fixed- or equal-length blocks. Because each record in a specific file is the same size, you can easily calculate the position of a record.

For instance, suppose you have a file with a record length of 50-bytes (or characters). To access Record 10, multiply the record number (10) by the record length (50} and move the file pointer to the calculated position (500).

A random access file sends and receives data (using PUT and GET) in a binary form, exactly as BASICO9 stores it internally. This feature minimizes the time involved in converting the data to and from ASCII representation, as well as reducing the file space required to store numeric data. You position the random access file pointer using SEEK. Compared to sequential file access, random file access using GET and PUT is very fast.

Using random access commands, you can store and retrieve individual bytes, strings of bytes, individual elements of arrays or total arrays with one PUT or GET command. When you GET a structure, you recover the number of bytes associated with that type of structure.

This means when you GET one element of byte type data, you read one byte. When you GET one element of real type data, you read five bytes. If you GET an array, you read all the elements of the array. This potential for reading entire arrays at once can greatly speed disk access.

As well as moving the file pointer to the beginning of individual records, you can also move it to any position within a record and begin reading or writing one or more bytes from that point.

Creating Random Access Files

You create and open random access files in the same way you create and open sequential files. The only differences are in the commands you use to store and retrieve the data and in the manner you keep track of where elements, or records, of a file begin and end.

Before you can write data to a random access file, you must either CREATE it or open it in the WRITE or UPDATE mode. Once you have a path open to an existing file, use PUT to write data into the file. If you open the file in the READ or UPDATE mode, you can then use the GET command to retrieve data from the file.

The PUT command can use only one parameter, the name of the data element to store. The parameter can be a string, a variable, an array, or a complex data structure,

Before storing data, you must devise a method to store it in blocks of equal size. Knowing the unit size lets you later retrieve the data in its original form. The following procedure shows one way to do this:

```
PROCEDURE putget
□REM This procedure creates a file named TEST1,
□REM reads 10 data lines, PUTs them into the file,
□REM then closes the file. Next it opens the file
□REM in READ mode, GETs the stored lines and lists
□REM them on the display screen.

□DIM LENGTH:BYTE
□DIM NULL:STRING[25]
□DIM LINE: STRING[25]
□DIM PATH:BYTE
□LENGTH=25
□NULL="□□□□□□□□□□□□□□□□□□□□□□□□□"
□BASE 0
□ON ERROR GOTO 10
□DELETE "TEST1" \ (* if the file exists, delete it.
□10 ON ERROR

□CREATE #PATH,"TEST1":WRITE \ (* create a file named TEST1.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T \ (* find beginning of each line.
□READ LINE \ (* read a line of data.
□PUT #PATH,LINE \ (* store the line in the file.
□NEXT T

□CLOSE #PATH \ (* close the file.
```

```
OPEN #PATH,"TEST1":READ \ (* open the file for reading.
FOR T=0 TO 9
SEEK #PATH,LENGTH*T \ (* find the beginning of each line.
GET #PATH,LINE \ (* get a line from the file.
PRINT LINE \ (* display the line.
NEXT T

CLOSE #PATH \ (* close the file.
END

DATA "This is test line #1."
DATA "This is test line #2."
DATA "This is test line #3."
DATA "This is fest line #4."
DATA "This is test line #5."
DATA "This is test line #6."
DATA "This is test line #7."
DATA "This is test line #8."
DATA "This is test line #9."
DATA "This is test line #10."
```

This procedure creates a file named TESTl. The variable named LENGTH stores the length of each line in the file (25 characters). The string variable NULL, is a string of 25 space characters. The variable LINE contains the data to store in each element (record) in the file. The variable PATH stores the path number of the file.

Next, the procedure contains an ON ERROR routine that deletes the file TESTl, if it already exists. Without this routine, the procedure produces an error if you execute it more than once.

Next, the routine uses CREATE to open the file TESTl. The line `SEEK #PATH, LENGTH*T` sets the file pointer to the proper location to store the next line. Because LENGTH is established as 25, the file lines are stored at 0, 25, 50, 75, and so on.

After the routine initializes storage space, it begins to store data by reading the procedure data lines one at a time, seeking the proper file location, and putting the data into the file. After storing all 10 lines, it closes the file.

The last part of the routine opens the new file, uses the same SEEK routine to position the file pointer, and reads the lines back, one at a time, to confirm that the store routine is successful.

The next short routine shows how you can use a procedure to read any line you select in the file, without reading any preceding lines:

```
PROCEDURE randomread
DIM LENGTH:BYTE
DIM LINE:STRING[25]
DIM SEEKLINE:BYTE
DIM PATH:BYTE
LENGTH=25
OPEN #PATH,"TEST1":READ \      (* open the file for reading.
LOOP
INPUT "Line number to display...",SEEKLINE \(* type a line to get.
EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN \(* test if record is valid.
ENDEXIT \                      (* exit loop if not.
SEEK #PATH, (SEEKLINE-1)*LENGTH \ (* find the requested record.
GET #PATH,LINE \               (* read the record.
PRINT LINE \                   (* display the record.
PRINT
ENDLOOP
PRINT "That's all ....... " \  (* end session.
CLOSE #PATH \                  (* close path.
END
```

The procedure asks for the record number of the line to display. When you type the number (1-10) and press [ENTER], SEEK moves the file pointer to the beginning of the record you want, GET reads it into the variable LINE, and PRINT displays it. The calculation (SEEKLINE-1)*LENGTH determines the beginning of the line you want. If you type a number outside the range of lines contained in the file (1-10), the procedure drops down to line 100 and ends.

By changing this procedure slightly, you can replace any line in the procedure with another line. The altered procedure below demonstrates this:

```
PROCEDURE random_replace
DIM LENGTH:BYTE
DIM LINE:STRING[25]
DIM SEEKLINE:BYTE
DIM PATH:BYTE
LENGTH=25
OPEN #PATH,"TEST1":UPDATE \     (* open the file.
LOOP
INPUT "Line number to display...",SEEKLINE \(* type record to
find.
EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN \(* test if valid number.
ENDEXIT \                       (* exit loop if not.
SEEK #PATH,(SEEKLINE-1)*LENGTH \(* find the requested record.
GET #PATH,LINE \                (* get the data.
PRINT LINE \                    (* print the record.
PRINT
```

```
INPUT "Type new Line... ",LINE \(* type a new line.
SEEK #PATH, (SEEKLINE-1)*LENGTH \(* find beginning of the record.
PUT #PATH,LINE \                  (* store the new line.
ENDLOOP \                         (* do it all again.
PRINT "That's all ....... " \     (* terminate procedure.
CLOSE #PATH \                     (* close path.
END
```

This time, the file is opened in the UPDATE mode to allow both reading and writing. You type the line you want to display. A prompt then asks you to type a new line. The procedure exchanges the new line for the original line, and stores it back in the file.

Using Arrays With Random Access Files

BASIC09's random access filing system is even more impressive when used with data structures, such as arrays. Instead of using a loop to store the 10 lines of the random_replace procedure, you could store them all at once, into one record, using an array. The following procedure illustrates this:

```
PROCEDURE arraywrite
DIM LENGTH:BYTE
DIM LINE:STRING[25]
DIM RECORD(10):STRING[25]
DIM PATH:BYTE
LENGTH=25

ON ERROR GOTO 10
DELETE "TEST2" \              (* delete TEST2 if it exists.
10 ON ERROR

CREATE #PATH,"TEST2":WRITE \ (* create TEST2.
BASE 0

FOR T=0 TO 9
READ RECORD(T) \              (* Read data lines into RECORD array.
NEXT T

SEEK #PATH,0 \                (* set pointer to beginning of file.
PUT #PATH,RECORD \            (*  store  the  entire  array  into  the
file.
CLOSE #PATH \                 (* close path to the file.

OPEN #PATH,"TEST2":READ \     (* open the file to read.
FOR T=0 TO 9
SEEK #PATH, LENGTH*T \        (* find each element.
GET #PATH,LINE \              (* read an element.
PRINT LINE \                  (* print the element.
NEXT T
```

```
CLOSE #PATH
END

DATA "This is test line #1"
DATA "This is test line #2"
DATA "This is test line #3"
DATA "This is test line #4"
DATA "This is test line #5"
DATA "This is test line #6"
DATA "This is test line #7"
DATA "This is test line #8"
DATA "This is test line #9"
DATA "This is test line #10"
```

This procedure reads the 10 lines into an array named RECORD. Then it places the entire array in the TEST1 file, using one PUT statement. To show that the structure of the file is still the same, the original FOR/NEXT loop reads the lines, one at a time, and displays them.

Notice that, because you need to write only one element, you can set the file pointer to 0 `(SEEK #PATH,0)`. You can rewind a file pointer (set it to 0) at any time in this manner.

You could save additional programming space by also reading the 10 lines back into memory as an array. The following procedure uses a new array, READLINES, to call the file back into memory, and displays the lines.

```
PROCEDURE arrayread
BASE 0
DIM READLINES(10):STRING[25]
DIM PATH:BYTE
OPEN #PATH,"TEST2":READ \    (* open file.
GET #PATH,READLINES \        (* read file into array.
CLOSE #PATH
FOR T=0 TO 9
PRINT READLINES(T) \         (* print each element of the array.
NEXT T
END
```

Using Complex Data Structures

In the previous section, you stored and retrieved elements of an array that were all the same size, 25 characters. Often you need to store elements of varying sizes, such as when you create a data base program with several fields in one record.

The following examples create a simple inventory system that requires a random access file having 100 records. Each record includes the name of the item (a 25-byte string), the item's list price and cost (both real numbers), and the quantity on hand (an integer).

First, you use the TYPE command to define a new data type that describes such a record. For example:

```
TYPE INV_ITEM=NAME:STRING[25]; L1ST,COST:REAL;
QTY: INTEGER
```

Although this statement describes a new record type called INV_ITEM, it does not assign variable storage for the record. The next step is to create two data structures: an array of 100 records of type INV_ITEM named INV_ARRAY and a working record named WORK_REC. The following lines do this:

```
DIM INV_ARRAY(188):INV_ITEM
DIM WORK_REC: INV_ITEM
```

To determine the number of bytes assigned for each type, you can use BASIC09's SIZE command. SIZE returns the number of bytes assigned to any variable, array, or complex data structure. For example, the instruction `SIZE(WORK_REC)` returns the number 37. The instruction `S1ZE(INV_ARRAY)` returns the number 3700.

You can use SIZE with SEEK to position a file pointer to a specific record's address.

The following procedure creates a file called inventory and immediately initializes it with zeroes and nulls strings. Five INPUT lines then ask you for a record number and the data to store in each field of the record. You can fill any record you choose, from 1 through 100,

When one record is complete, the procedure uses PUT to store the record. Then, it asks you for a new record number. If you wish to quit, enter a number either larger than 100 or smaller than 1.

```
PROCEDURE inventory
REM Create a data type consisting of a 25-character
REM name field, a real list price field, a real cost field,
REM and an integer quantity field.

TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY: INTEGER
DIM INV_ARRAY(100):INV_ITEM \     (* dimension an array using new
type.
DIM WORK_REC:INV_ITEM
REM dimension a working variable of the new type.
DIM PATH:BYTE

ON ERROR GOTO 10
DELETE "inventory"
10 ON ERROR

CREATE #PATH,"inventory" \  (* create a file named inventory.
WORK_REC.NAME ="" \         (* set all data elements to null or 0.
WORK_REC.LIST=0
WORK_REC.COST=0
WORK_REC.QTY=0
FOR N=1 TO 100
PUT #PATH,WORK_REC
NEXT N

LOOP
INPUT "Record number? ",RNUM \    (* enter number of record to
write.
IF RNUM<1 OR RNUM>100 THEN \      (* check if number is valid.
PRINT
PRINT "End of Session" \          (* if not, end session.
PRINT
CLOSE #PATH
END
ENDIF
INPUT "Item name? ",WORK_REC.NAME \(* type data for record.
INPUT "List price? ",WORK_REC.LIST
INPUT "Cost price? ",WORK_REC.COST
INPUT "Quantity? ",WORK_REC.QTY
SEEK #PATH, (RNUM-1)*SIZE(WORK_REC) \ (* find record.
PUT #PATH,WORK_REC \                  (* write record to file.
ENDLOOP
```

Notice that the INPUT statements reference each field separately, but the PUT statement references the record as a whole.

The next procedure lets you read any record in your inventory file, and displays that record. If you ask for a record you have not yet filled with meaningful data, the display consists of a null string and zeroes.

```
PROCEDURE readinv
TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
DIM WORK_REC:INV_ITEM

DIM PATH:BYTE
OPEN #PATH, "inventory":READ
LOOP
INPUT "Record number to display? ",RNUM
IF RNUM<1 OR RNUM>100 THEN
PRINT "End of Session"
PRINT
CLOSE #PATH
END
ENDIF
SEEK #PATH, (RNUM-1)*SIZE(WORK_REC)
GET #PATH,WORK_REC
PRINT "#","Item","List Price","Cost Price","Quantity"
PRINT
"-------------------------------------------------------------------
--------"
PRINT RNUM, WORK_REC.NAME, WORK_REC.LIST, WORK_REC.COST,
WORK_REC.QTY
PRINT
ENDLOOP
END
```

This procedure accesses the file one record at a time. It is not necessary to do so. You can read the entire file into memory at once by dimensioning an inventory array and getting the whole file into it:

```
TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
DIM INV_ARRAY(100):INV_ITEM
SEEK #PATH,0 \ (* rewind the file.
GET #PATH, INV_ARRAY
```

The examples in this section are simple, yet they illustrate the combined power of BASIC09 complex data structures and the random access file statements. They show that a single GET or PUT statement can move any amount of data, organized in any way you want. Other advantages of using complex data structures are:

- The procedures are self-documenting. You can see easily what a procedure does because its structures can have descriptive names.
- Execution is extremely fast.

- Procedures are simple and usually require fewer statements to perform IO functions than other BASICs.
- The procedures are versatile. By creating appropriate data structures, you can read or write almost any kind of data from any file, including files created by other programs or languages.

Chapter 9

Displaying Text and Graphics

BASIC09 has three levels of graphics capabilities. The first (low resolution) and third (high resolution) levels can include both graphics designs and text. The second (medium resolution) level can display only graphics designs.

NitrOS-9/EOU introduced new versions of the Inkey, SysCall, GFX and GFX2 subroutine modules. They replace the first character of the module name with the letter **B**. These versions are merged with the BASIC09 module and are intended for use during application development. They can be used until your procedure(s) reach 32K of workspace. After that, you will have to rename your RUN statements for these subroutines back to their original names and begin packing the procedures to use with RunB. The originally named subroutines are merged with RunB, so when you pack the procedures you can use them with RunB.

Examples:

```
RUN BFX("CLEAR")
RUN BFX2("GCOLR",xcor,ycor,color)
RUN BNKEY(key)
RUN BYSCALL(callCode,regs)
```

To change all occurrences of these to their original names:

```
c*/BFX/GFX/
c*/BFX2/GFX2/
c*/BNKEY/INKEY/
c*/BYSCALL/SYSCALL/
```

You *must* return to the top of the procedure (using the `-*` command) for each of these commands in order to ensure all occurrences of each are properly changed. You only need to use the commands for the subroutines you used in a given procedure.

ASCII Codes

There are some keys which do not exist on a Color Computer keyboard. To remedy this, Microware created some special key-mappings. They are listed here for your convenience, as some of them ([,], and \) are used in BASIC09.

	ASCII Value		Keys to Press	
Character	**Decimal**	**Hexadecimal**	**Level One**	**Level Two**
_	95	5F	[CLEAR] [-]	[CTRL] [-]
{	123	7B	[CLEAR] [,]	[CTRL] [,]
}	125	7D	[CLEAR] [.]	[CTRL] [.]
\	92	5C	[CLEAR] [/]	[CTRL] [/]
|	124	7C	[CLEAR] [1]	[CTRL] [1]
~	126	7E	[CLEAR] [3]	[CTRL] [3]
^	94	5E	[CLEAR] [7]	[CTRL] [7]
[	91	5B	[CLEAR] [8]	[CTRL] [8]
]	93	5D	[CLEAR] [9]	[CTRL] [9]
`	96	60		[SHIFT] [@]
carriage-return	13	0D	[ENTER]	[ENTER]
backspace	8	08	[←]	[←]
cursor right	9	09	[→]	[→]
cursor down	10	0A	[↓]	[↓]
cursor up	12	0C	[↑]	[↑]
clear screen				

The back-tick, [SHIFT][@]. will show as a forward tick on a hardware text screen, but properly as a backward tick on fonts that support it (including the standard Tandy/Microware Group 200, font 1).

For low-resolution text screens and high-resolution text and graphic screens, BASIC09 uses ASCII (American Standard Code for Information Interchange) codes to represent the common alphanumeric characters. ASCII is the same code that most small computers use.

There are times when you need to know the ASCII value of a printable character. A table of the standard codes follows, showing decimal and hexadecimal values for the common alphanumeric characters:

	ASCII Value			ASCII Value	
Character	**Decimal**	**Hexadecimal**	**Character**	**Decimal**	**Hexadecimal**
Space	32	20	**K**	75	4B
!	33	21	**L**	76	4C
"	34	22	**M**	77	4D
#	35	23	**N**	78	4E
$	36	24	**0**	79	4F
%	37	25	**P**	80	50
&	38	26	**Q**	81	51
'	39	27	**R**	82	52
(	40	28	**S**	83	53
)	41	29	**T**	84	54
*	42	2A	**U**	85	55
+	43	2B	**V**	86	56
,	44	2C	**W**	87	57
-	45	2D	**X**	88	58
.	46	2E	**Y**	89	59
/	47	2F	**Z**	90	5A
0	48	30	**a**	97	61
1	49	31	**b**	98	62
2	50	32	**c**	99	63
3	51	33	**d**	100	64
4	52	34	**e**	101	65
5	53	35	**f**	102	66
6	54	36	**g**	103	67
7	55	37	**h**	104	68
8	56	38	**i**	105	69
9	57	39	**j**	106	6A
:	58	3A	**k**	107	6B
;	59	3B	**1**	108	6C
<	60	3C	**m**	109	6D
=	61	3D	**n**	110	6E
>	62	3E	**o**	111	6F
?	63	3F	**p**	112	70
@	64	40	**q**	113	71
A	65	41	**r**	114	72
B	66	42	**s**	115	73

C	67	43	**t**	116	74
D	68	44	**u**	117	75
E	69	45	**v**	118	76
F	70	46	**w**	119	77
G	71	47	**x**	120	78
H	72	48	**y**	121	79
I	73	49	**z**	122	7A
J	74	4A			

You can use the CHR$ function with the ASCII values of these characters to produce the displayable character.

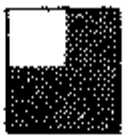
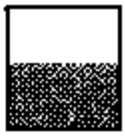
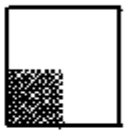
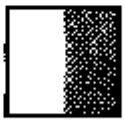
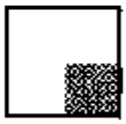
Low-Resolution Graphic Characters

In addition to alphanumeric characters, low-resolution graphics also offers graphic characters. Generate these characters by pressing [ALT] at the same time you press a keyboard character. The graphics character codes are in the range 128-255.

Pressing [ALT] while pressing another key, causes NitrOS-9 to add 128 to the ASCII value of the second key. (For the technically minded, NitrOS-9 sets the high bit of the character code.) Therefore, if you press [ALT][A], you produce graphics character 193. You can also generate graphics characters from BASIC09 using the CHRS$ function, and you can PRINT them in the same manner as other characters.

Low-level graphics characters follow a pattern that repeats every 16 characters. Table 9.1 shows the first set of graphic characters, 128-143.

Table 9.1
Low-Resolution Graphic Character Set

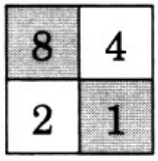
Character Code	Character Code	Character Code	Character Code
 128	 132	 136	 140
 129	 133	 137	 141
 130	 134	 138	 142
 131	 135	 139	 143

Subsequent characters produce the same series of configurations but display in different colors, as shown in Table 9.2.

Table 9.2
Low-Resolution Graphic Color Set

ASCII Code	Graphics Block Color
128 - 143	Black and Green
144 - 159	Black and Yellow
160 - 175	Black and Blue
176 - 191	Black and Red
192 - 207	Black and Buff
208 - 223	Black and Light Blue
224 - 239	Black and Cyan
240 - 254	Black and Orange
255	Green

Within each color set, you can easily calculate the number for a particular character. For instance, suppose you want to print a character that has orange upper left and lower right corners. Picture the character divided into four sections, numbered as follows:

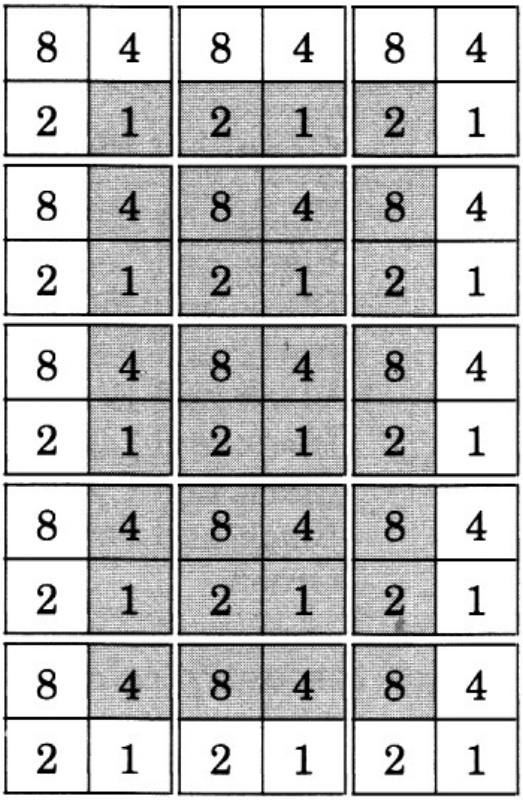


To calculate a character that has orange at Sections 8 and 1, add the section values to the first value in the orange group, 240, like this:

```
240 + 8 + 1 = 249
```

Character 249 is what you want.

The following diagram shows how you might block out a large letter O on the screen. The shaded portions of the characters are colored. The unshaded portions are black. In this case we want the colored portions to be green (the same color as the screen). You can do this using the color set 128 - 143.



Because Section 1 in the upper left character is to be colored, add 1 to the initial character value of 128. The first character value is 129. Moving right, Sections 2 and 1 are colored in the second character. Add 3 to 128 to get a second character value of 131. Calculate all 15 characters in this manner.

You could create a letter O in a BASIC09 procedure by printing each of the five rows of three characters. You could use DATA lines to store the ASCII codes for each character, then use loops to read and display the characters they represent.

Although low-level graphics is very rough, it can be useful, and it lets you mix graphics with text.

The following procedure not only creates the letter O, it adds the letter S and the number 9 to display the name of your operating system.

```
PROCEDURE os9prog
DIM DAT:INTEGER
PRINT CHR$(12)
PRINT
PRINT
PRINT
FOR Z=1 TO 5
PRINT TAB(10);
FOR T=1 TO 12
READ DAT
PRINT CHR$(DAT);
NEXT T
PRINT
NEXT Z
END
DATA 129,131,130,143,129,131,131,143,129,131, 130,143
DATA 133,143,138,143,133,143,143,143,132,140, 136,143
DATA 133,143,138,143,132,140,140,143,131,131,  130,143
DATA 133,143,138,143,131,131,130,143,143,143, 138,143
DATA 132,140,136,143,140,140,136,143,143,143, 138,143
```

Special Characters in High-Resolution

High-resolution graphics does not have graphic characters but it does have other international and special characters. These characters are represented by ASCII codes 128 through 159 as shown in the following table:

Table 9.3
High-Resolution Special Characters

Character	Hex Code	Decimal Code	Keypress	Character	Hex Code	Decimal Code	Keypress
Ç	80	128	[ALT][A]	**ó**	90	144	[ALT][Q]
ü	81	129	[ALT][B]	**æ**	91	145	[ALT][R]
é	82	130	[ALT][C]	**Æ**	92	146	[ALT][S]
å	83	131	[ALT][D]	**ô**	93	147	[ALT][T]
ä	84	132	[ALT][E]	**ö**	94	148	[ALT][U]
à	85	133	[ALT][F]	**ø**	95	149	[ALT][V]
ā	86	134	[ALT][G]	**û**	96	150	[ALT][W]
ç	87	135	[ALT][H]	**ù**	97	151	[ALT][X]
ê	88	136	[ALT][I]	**Ø**	98	152	[ALT][Y]
ë	89	137	[ALT][J]	**Ŏ**	99	153	[ALT][Z]
è	8A	138	[ALT][K]	**Ü**	9A	154	[ALT][:]
ï	8B	139	[ALT][L]	**§**	9B	155	[ALT][;]
î	8C	140	[ALT][M]	**£**	9C	156	[ALT][,]
ß	8D	141	[ALT][N]	±	9D	157	[ALT][=]
Ä	8E	142	[ALT][O]	°	9E	158	[ALT][.]
Å	8F	143	[ALT][P]	***ƒ***	9F	159	[ALT][/]

Medium-Resolution Graphics

For more sophisticated graphics operations, NitrOS-9 has built-in graphics interface modules that provide a convenient way to access the graphics and joystick functions of the Color Computer 3. The required module for medium-resolution graphics is named GFX. It must be in your execution directory or resident in memory when called by BASIC09. The GFX module is for Level One functions, and requires running on a VDG style (32x16) window. (JOYSTK is a special case in that it will work on any type of window.)

You can either install GFX in memory using the LOAD command, or wait until BASIC09 calls it for a graphics function. Once loaded, GFX resides in memory until you remove it using the NitrOS-9 UNLINK command or the BASIC09 KILL command.

GFX has a number of functions that you pass to it as parameters with the RUN statement. For instance, the following statement clears the current graphics screen:

```
RUN GFX('CLEAR")
```

Other tasks need such parameters as position, color, and size. The following is a quick reference to all of the GFX functions. Each is explained in detail later:

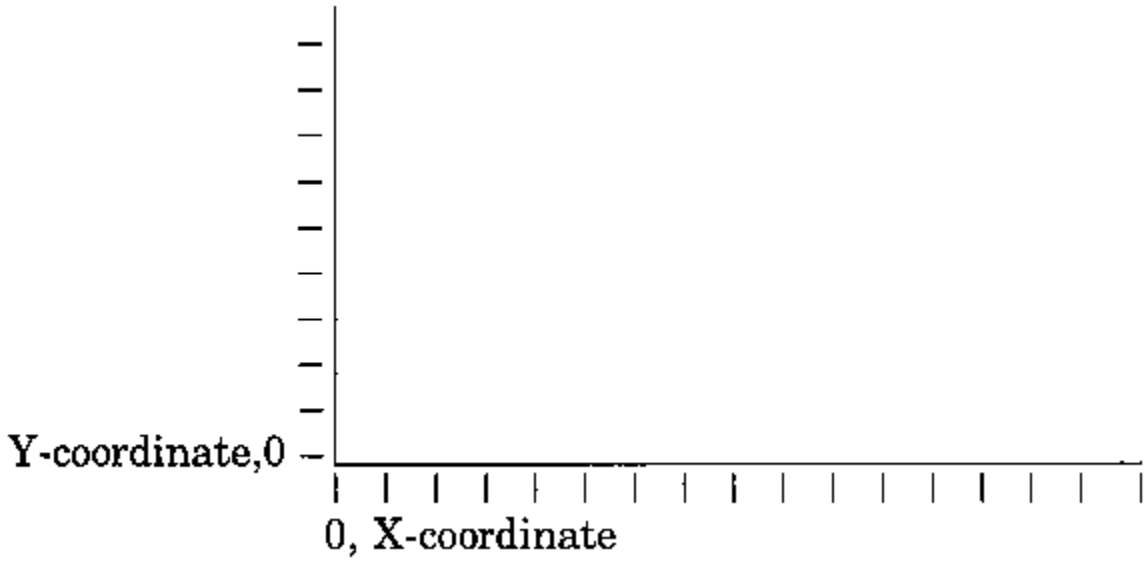
Function	Purpose	Parameters
ALPHA	Sets the screen to the alphanumeric mode.	None.
CIRCLE	Draws a circle.	Radius, optional X- and Y-coordinates, and color.
CLEAR	Clears the screen to a color.	Optional color for screen.
COLOR	Changes the foreground and background colors.	Foreground and background colors.
FILL	Fills (paints) foreground color starting at current Graphics cursor address. Fills any adjacent pixels that are the same color as the pixel the Fill starts at.	None
GCOLR	Reads a pixel's color.	Names of variables in which to store optional X- and Y-coordinates.
GLOC	Returns a video display address.	INTEGER variable that it returns the screen address in.
JOYSTK	Returns the joystick button and X- and Y-coordinate status.	Names of variables in which to return the values.
LINE	Draws a line.	Ending X- and Y-coordinates, optional beginning coordinates, optional color.
MODE	Switches the screen between alphanumeric and graphics, sets the graphics screen color.	Format, Color,
MOVE	Positions the invisible graphics cursor.	X- and Y-coordinates.
POINT	Moves graphics cursor and sets a point.	X- and Y-coordinates and optional pixel color.
QUIT	Returns screen to alphanumeric mode. Deallocates graphics memory.	None.

Formats and Colors

In medium-resolution graphics, you have a choice of two formats. Format 0 provides 256 horizontal points by 192 vertical points. In this format, you can have only two colors on the screen at a time.

Format 1 provides a 128 by 192 point screen and a maximum of four colors on the screen at a time. NitrOS-9 medium-resolution graphics treats the screen as if it

were a grid, with coordinate 0,0 at the lower left corner as shown in the following illustration. All points on the grid are positive.



BASIC09 defines colors with numbers or color codes. Many GFX functions allow or require color codes as parameters. BASIC09 also divides the color codes into color sets. Specifying a color code outside the current color set automatically initializes the new set.

Color Set	Color Code	Format 0		Color Code	Format 1	
		Background	Foreground		Background	Foreground
1	00	Black	Black	00	Green	Green
	01	Black	Green	01	Green	Yellow
	02			02	Green	Blue
	03			03	Green	Red
2	04	Black	Black	04	Buff	Buff
	05	Black	Buff	05	Buff	Cyan
	06			06	Buff	Magenta
	07			07	Buff	Orange
3				08	Black	Black
				09	Black	Dk Green
				10	Black	Md Green
				11	Black	Lt Green
4				12	Black	Black
				13	Black	Green
				14	Black	Red
				15	Black	Buff

Table 9.4

Use the preceding charts to choose colors for those functions that let you specify foreground or background colors. For instance, to initialize a Format 1 graphics screen with a green background and a red foreground, you type:

```
run gfx("mode",1,3)
```

The following reference section describes all the medium-resolution graphics functions, and provides examples and sample programs. To understand the organization of the commands reference, see "The Syntax Line" in Chapter 11.

The Draw Pointer

Medium-resolution graphics uses a draw pointer, or invisible graphics cursor, to determine what area of the screen is affected by graphics operations. When you establish a graphics screen, the draw pointer is located at coordinates 0,0. Some graphic functions automatically change the pointer location on the screen, For instance, the LINE function moves the draw pointer from the beginning coordinates to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the MOVE function to set the draw pointer to new locations.

ALPHA Select alphanumeric screen

Syntax: RUN GFX("ALPHA")

Function: Switches from the graphics screen to the alphanumeric (text) screen. The current graphics screen remains intact.

Parameters:

None

Examples:

```
RUN GFX("ALPHA")
```

Sample Procedure:

This procedure lets you choose to draw a circle or rectangle of the size you select. Once you choose the shape and size, it uses the MODE function to select a graphics screen. When the shape is complete, you press [ENTER] to return to a text screen. The procedure uses the ALPHA function to return to the original menu.

```
PROCEDURE alpha
DIM CLS:BYTE
DIM XCOR,YCOR,SIDE1,SIDE2,RADIUS,T,X,Y,Z:INTEGER
DIM RESPONSE:STRING[1]
CLS=12
10 REPEAT
PUT #1,CLS
PRINT "Do you want to draw"
PRINT "1) A rectangle"
PRINT "2) A circle"
PRINT "3) Quit"
PRINT "   -Press 1, 2 or 3...";
GET #0,RESPONSE
PRINT
IF RESPONSE="1" THEN
INPUT "Length of Side 1",SIDE1
INPUT "Length of Side 2",SIDE2
RUN GFX("MODE",0,0)
RUN GFX("CLEAR")
XCOR=10
YCOR=10
RUN GFX("LINE",XCOR,YCOR,XCOR+SIDE1,YCOR,1)
RUN GFX("LINE",XCOR+SIDE1,YCOR,XCOR+SIDE1,YCOR+SIDE2,1)
RUN GFX("LINE",XCOR+SIDE1,YCOR+SIDE2,XCOR,YCOR+SIDE2,1)
RUN GFX("LINE",XCOR,YCOR+SIDE2,XCOR,YCOR,1)
INPUT RESPONSE
ELSE
IF RESPONSE="2" THEN
```

```
INPUT "What radius?...",RADIUS
RUN GFX("MODE",0,1)
RUN GFX("CLEAR")
RUN GFX("CIRCLE",128,90,RADIUS)
INPUT RESPONSE
ENDIF
ENDIF
UNTIL RESPONSE<>"1" AND RESPONSE<>"2"
RUN GFX("ALPHA")
IF RESPONSE<>"3" THEN 10
END
```

CIRCLE Draw a circle

Syntax: RUN GFX("CIRCLE"[,xcor,ycor],radius [,color])

Function: Draws a circle of a given radius. If you do not specify a color, BASIC09 uses the current foreground color. If you do not specify X- and Y-coordinates, CIRCLE uses the current graphics cursor position as the circle's center.

Parameters:

radius	The radius of the circle you want to draw.
color	The code of the color you want the circle to be. See the chart earlier in this section for color information.
xcor,ycor	The X- and Y-coordinates for the center of the circle. Specifying coordinates outside the X-coordinate range of 0-255 or outside the Y-coordinate range of 0-199 causes an error.

Examples:

```
RUN GFXC"CIRCLE",100)
RUN GFX("CIRCLE",100,3)
RUN GFXC"CIRCLE",125,100,100)
RUN GFX("CIRCLE",125,100,100,2)
```

Sample Procedure:

This procedure uses CIRCLE to draw and erase a circle. The location of the circle changes before each draw/erase operation, causing the circle to move. When it hits the edge of the screen, it reverses its direction at a random angle and *bounces*.

```
PROCEDURE circles
DIM RADIUS,XCOR,YCOR:INTEGER
DIM XTEMP,YTEMP:INTEGER
DIM PATH1,PATH2:INTEGER
DIM FLAG:INTEGER
FLAG=1
XCOR=5
YCOR=5
PATH1=RND(15)+2
PATH2=RND(10)+2
XTEMP=249
YTEMP=185
RUN GFX("MODE",0,1)
RUN GFX("CLEAR")
FOR T=1 TO 200
WHILE XCOR<250 AND XCOR>4 AND YCOR<186 AND YCOR>4
 DO
RUN GFX("CIRCLE",XTEMP,YTEMP,3,0)
```

```
RUN GFX("CIRCLE",XCOR,YCOR,3,1)
XTEMP=XCOR
YTEMP=YCOR
XCOR=XCOR+PATH1
YCOR=YCOR+PATH2
ENDWHILE
PATH1=RND(15)+2
PATH2=RND(10)+2
IF XCOR>249 THEN
XCOR=249
ENDIF
IF XCOR<5 THEN
XCOR=5
ENDIF
IF YCOR>185 THEN
YCOR=185
ENDIF
IF YCOR<5 THEN
YCOR=5
ENDIF
FLAG=FLAG*-1
IF FLAG<0 THEN
PATH1=PATH1*-1
PATH2=PATH2*-1
ENDIF
NEXT T
END
```

CLEAR Clear the screen

Syntax: RUN GFX("CLEAR"[,color])

Function: Clears the current graphics screen. If you do not specify a color, CLEAR sets the entire screen to the current background color. CLEAR also sets the graphics cursor at coordinates 0,0, the upper left corner of the screen.

Parameters:

color A code indicating the color to set the screen.

Examples:

```
RUN GFX("CLEAR")
RUN GFX("CLEAR",14)
```

COLOR Change the foreground color

Syntax: RUN GFX("COLOR",color)

Function: Changes the foreground color (and possibly the color set). COLOR does not change the graphics format or the cursor position.

Parameters:

color	A code indicating the color you want for the foreground. See the chart earlier in this chapter for color information.

Examples:

```
RUN GFX("COLOR",10)
```

Sample Procedure:

This procedure connects a series of differently colored circles to produce a necklace effect.

```
PROCEDURE necklace
DIM COLOR,T,U,J,R,FLAG,XCOR,YCOR:INTEGER
RUN GFX("MODE",1,0)
RUN GFX("CLEAR")
COLOR=1
XCOR=1
YCOR=1
R=2
FLAG=1
FOR T=1 TO 6
FOR J=1 TO 40
XCOR=XCOR+1
YCOR=YCOR+.8
IF FLAG<0 THEN
R=R-1
ELSE
R=R+1
ENDIF
COLOR=COLOR+1
IF COLOR>3 THEN COLOR=1
ENDIF
RUN GFX("CIRCLE",XCOR,YCOR,R,COLOR)
NEXT J
FLAG=FLAG*-1
NEXT T
FOR U=1 TO 10000
NEXT U
END
```

FILL — Flood fill with current foreground color at current graphics cursor location

Syntax: **RUN GFX("FILL")**

Function: Will fill in adjacent pixels that are the same color as the pixel the fill started from.

Parameters:

None

Examples:

```
RUN GFX("FILL")
```

GCOLR Read the color of a pixel

Syntax: RUN GFX("GCOLR"[,xcor,ycor],color)

Function: Read the color of a pixel at the current graphics cursor location, or from the coordinates xcor,ycor.

Parameters:

xcor,ycor	Xcor and ycor are optional coordinates to read the color value from.
color	Color is the color value of the pixel read, may a byte or an integer.

Examples:

```
RUN GFX("GCOLR",color)
RUN GFX("GCOLR",xcor,ycor,color)
```

GLOC Find the graphics screen location

Syntax: RUN GFX("GLOC",storage)

Function: Determines the location of the graphics screen in memory and returns the address in the specified variable. When you know the graphic screen address, you can use PEEK and POKE to perform special functions not available in the GFX module, such as filling a portion of the screen with a color or saving a graphics screen to disk.

NitrOS-9 Level Two maps display screens into a procedure's address space before PEEK and POKE can operate on a display screen. This means that you must have at least eight kilobytes of free memory in the user's address space. Procedure and data memory requirements must not exceed 56 kilobytes.

Parameters:

storage An integer type variable in which GLOC stores the memory address that the screen is mapped in at in the current process (assuming a VDG
text or Coco 1/2 graphics screen).

Examples:

```
RUN GFX("GLOC",location)
```

Sample Procedure:

This procedure uses the GLOC function to locate the current graphics screen, then uses POKE to paint a series of boxes on the screen.

```
PROCEDURE boxin
DIM LOCATION,PLACE,COLOR,BEGIN,QUIT,X,TERMINATE,
LINE,T,J:INTEGER
RUN GFX("MODE",1,0)
RUN GFX("CLEAR")
RUN GFX("GLOC",LOCATION)
LOCATION=LOCATION + 100 \ PLACE=LOCATION
BEGIN=1
QUIT=80
COLOR=255
TERMINATE=10
LINE=32
FOR x=1 TO 4
FOR T=1 TO QUIT
FOR J=BEGIN TO TERMINATE
POKE PLACE+J,COLOR
NEXT J
PLACE=PLACE+LINE
```

```
    NEXT T
    LOCATION=LOCATION+160
    BEGIN=BEGIN+1
    PLACE=LOCATION
    QUIT=QUIT-10
    TERMINATE=TERMINATE-1
    COLOR=COLOR-85
    NEXT X
    INPUT Z$
    END
```

JOYSTK Get joystick status

Syntax: RUN GFX("JOYSTK",stick,fire,xcor,ycor)

Function: Determines the status of the specified joystick fire button and the X,Y position of the specified joystick handle. Use this function only with a standard joystick or mouse, not with the high-resolution mouse adapter. If you want to make a procedure that will be single button driven (and will work under both Level One and Level Two), you should check the button status for 0 (no buttons pressed) or <>0 (at least one button pressed).

Parameters:

stick The joystick (0 or 1) for which you want to determine the status.
0 indicates the right joystick, 1 indicates the left joystick.

fire A variable in which JOYSTK returns the status of the specified fire button. Fire can be byte, integer, or boolean type. Under Level One, a value of $FF or TRUE indicates the button is pressed. Under Level Two, it can return values of 1,2 or 3, with the following meanings:

- 1=Button 1 pressed
- 2=Button 2 pressed
- 3=Both buttons pressed

xcor,ycor Byte or integer type variables in which JOYSTK stores the X- and Y-coordinates of the joystick handle position. The coordinate range is 0-63.

Examples:

```
RUN GFX("JOYSTK",0,shoot,x,y)
```

Sample Procedure:

This procedure uses the JOYSTK function to draw on the screen with the right joystick.

```
PROCEDURE joydraw
  DIM STICK,FIRE,XCOR,YCOR,XTEMP,YTEMP:INTEGER
  RUN GFX("MODE",2,1)
  RUN GFX("CLEAR")
  XCOR=0 \YCOR=0
  REPEAT
  XTEMP=XCOR
  YTEMP=YCOR
```

```
RUN GFX("JOYSTK",0,FIRE,XCOR,YCOR)
XCOR=XCOR*4
YCOR=YCOR*4
RUN GFX("LINE",XTEMP,YTEMP,XCOR,YCOR)
UNTIL FIRE<>0
END
```

LINE Draw a line

Syntax: RUN GFX("LINE"[,xcor1,ycor1],xcor2 ycor2[,color])

Function: Draws a line in the current or specified foreground color in one of the following ways:

- From the current draw position to the specified X,Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X,Y-coordinates.

Parameters:

xcor1,ycor1	Are LINE's beginning X- and Y-coordinates.
xcor2,ycor2	Are LINE's ending X- and Y-coordinates.
color	A code indicating the color you want the line to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("LINE",192,128)
RUN GFX("LINE",0,0,192,128)
RUN GFX("LINE",0,0,192,128,2)
```

Sample Procedure:

This procedure draws a sine wave of vertical lines across the screen.

```
PROCEDURE waves
DIM A,B,C,D,X,Y,Z: INTEGER
CALC=0 \ A=100
RUN GFX("MODE",0,1)
RUN GFX("CLEAR")
RUN GFX("COLOR",2)
FOR X=0 TO 255 STEP 1
CALC=CALC+.05
Y=A-SIN(CALC)*15
Z=Y+25
RUN GFX("LINE",X,Y,X,Z)
NEXT X
END
```

MODE Switch to graphics screen

Syntax: RUN GFX("MODE",format,color)

Function: Switches the screen from alphanumeric (text) to graphics, selecting the screen format and color code. You must run MODE before you can use any other graphics function. When you do, BASIC09 allocates a six-kilobyte block of memory for graphics. If your system does not have this amount of memory available, NitrOS-9 returns an error message.

Parameters:

format	Either 0 (a two-color 256 by 192 pixel screen) or 1 (a four-color, 128 by 192 pixel screen).
color	A code indicating the color to set the screen. See the chart earlier in this chapter for information on color sets.

Examples:

```
RUN GFX("MODE",1,2)
```

MOVE Move graphics cursor

Syntax: RUN GFX("MOVE",xcor,ycor)

Function: Moves the invisible graphics cursor to the specified location on the screen. MOVE does not change the display in any way.

Parameters:

xcor,ycor The coordinates for the cursor.

Examples:

```
RUN GFX("MOVE",192,128)
```

Sample Procedure:

This procedure draws and pops bubbles on the screen using the CIRCLE function. It uses MOVE to select the position for the circles.

```
PROCEDURE bubbles
DIM XCOR,YCOR,T,R,ARRAY(3,100):INTEGER
RUN GFX("MODE",1,0)
RUN GFX("CLEAR")
FOR T=1 TO 20
ARRAY(1,T)=RND(255)
ARRAY(2,T)=RND(192)
ARRAY(3,T)=RND(50)
RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
RUN GFX("CIRCLE",ARRAY(3,T),3)
NEXT T
FOR T=1 TO 28
RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
RUN GFX("CIRCLE",ARRAY(3,T),3)
SHELL "DISPLAY 07"
NEXT T
END
```

POINT Set point to specified color

Syntax: RUN GFX("POINT",xcor,ycor[,color])

Function: Displays a dot at the specified coordinates. If you specify a color, POINT sets the pixel at the new coordinates to that color. Otherwise, POINT sets the pixel at the new coordinates to the foreground color.

Parameters:

xcor,ycor	The coordinates for a pixel.
color	The code of the color you want the pixel to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("POINT",192,128)
RUN GFX("POINT",192,128,2)
```

Sample Procedure:

```
PROCEDURE boxup
DIM XCOR,YCOR,BEGIN,COLOR,QUIT,TERMINATE,
 LINE:INTEGER
DIM T,X,Y:INTEGER
XCOR=50 \YCOR=30 \COLOR=1
BEGIN=1 \START=1 \QUIT=20 \TERMINATE=50
RUN GFX("MODE",1,0)
RUN GFX("CLEAR")
FOR T=1 TO 4
FOR X=BEGIN TO QUIT
FOR Y=START TO TERMINATE
RUN GFX("POINT",XCOR+Y,YCOR,COLOR)
NEXT Y
YCOR=YCOR+1
NEXT X
START=START+10
TERMINATE=TERMINATE-10
COLOR=COLOR+1
NEXT T
INPUT Z$
END
```

QUIT Deallocate graphics screen

Syntax: RUN GFX("QUIT")

Function: Switches the screen to the alphanumeric (text) mode and deallocates graphics memory.

Parameters:

None

Examples:

```
RUN GFX("QUIT")
```

High-Resolution Graphics

BASIC09's high-resolution graphics greatly expand the capabilities of the Color Computer 3. You can have greater screen resolution (up to 640 by 200 pixels), as many as 64 colors, and the ability to mix graphics and text on one screen. In addition, you can use different text fonts, or styles.

All drawing commands are faster if the text and graphics cursors are OFF at the time they are executed. So if you want to draw a screen using drawing commands and then have a mouse cursor and/or text cursor for interacting with the user, shut the cursors off, do the main drawing, and then turn them back on.

The high-resolution module, GFX2, has many more functions than its medium resolution counterpart. GFX2 gives you the ability to:

- Select from 64 colors. NitrOS-9 provides a palette with 16 default colors. You can change any of these default colors to any of the 64 colors available on the Color Computer 3.
- Set border colors.
- Set color patterns.
- Create different types of graphics screen cursors.
- Use logic functions.
- Turn an automatic scaling function off or on.
- Draw outline or filled boxes.
- Draw ellipses and arcs.
- Fill specified areas with specified colors.
- GET and PUT sections of the graphics screen.
- Select character fonts, which include boldfaced, transparent, and proportionally spaced characters.
- Move the cursor. Erase portions of a line or of the screen.
- Select reverse or normal video.
- Underline text.

Also, high-resolution graphics operate through the NitrOS-9 Windowing System. This means that you can run several procedures in different windows. You can establish windows to display text, or to display graphics, or both. You can easily display any window.

Establishing a Hardware Window

For your convenience, NitrOS-9 has a number of predefined or hardware window formats. Hardware windows are text windows, and you cannot use them for graphic applications. Because hardware windows are predefined, you can easily establish them with the INIZ command. For instance, to establish Window 7, type:

```
iniz /w7[ENTER]
```

However, you cannot see the window until you send a message to it. Type:

```
echo Hello Window 7 > /w7[ENTER]
```

Now, to see the window and your message press `[CLEAR]`. To return to the original screen, press `[CLEAR]` again.

To NitrOS-9, a window is a device and you can send data to it. To view the Errmsg file in the SYS directory of your system diskette, list it to Window 7 by typing:

```
list sys/errmsg > /w7[ENTER]
```

to move to Window 7 and see the listing. Press [SHIFT][CLEAR] to return to the previous screen.

You can also fork a shell (an execution environment) to a window. To cause a shell to operate in window 7, type:

```
shell i=/w7&[ENTER]
```

The i= function of SHELL tells NitrOS-9 that the window is *immortal*. It does not die after completing a task. To operate NitrOS-9 from the window, press `[CLEAR]`.

Besides window 7, you have six other predefined windows. The following chart shows all the hardware windows and their parameters:

Window Number	Screen Size Chars/line	Starting Coordinates X-Coord, Y-Coord	Window Size Cols	Rows
1	80	0,0	80	25
2	80	0,0	80	25
3	80	0,0	80	25
4	80	0,0	80	25
5	80	0,0	80	25

6	80	0,0	80	25
7	80	0,0	80	25

The /TERM window can be set in /dd/sys/env.file, by changing the settings for CONDVTYP, CONXSIZ, CONYSIZ, CONFCLR, CONBCLR, and CONBDCLR, and this will change /TERM's setting as NitrOS-9 boots.

Defining Windows

As well as hardware windows, NitrOS-9 also lets you establish windows to your own specifications. You can set definable windows for either text or graphics, or both. You can locate them anywhere on a screen, and you can make them any size.

You initialize definable windows in the same manner you initialize hardware windows, using INIZ. If you want to have text on the window, you must merge SYS/Stdfonts (found on your system diskette) with the window (NitrOS-9/EOU pre-loads these fonts for you on boot). You can also establish a shell in a definable window, from which you can use NitrOS-9 or BASIC09.

To establish definable windows you must supply NitrOS-9 with information about the type of window you want (its graphic format), its size, and its location on the screen. The easiest way to do this is with the NitrOS-9 WCREATE command.

WCREATE requires a window format code in the form -s=*format code* to tell NitrOS-9 what type of a window you want. The following chart shows the possible window formats you can choose:

Table 9.5

Format Code	Screen Size Cols x Rows	Resolution Width/Height	No. of Colors	Memory Required	Screen Type
01	40 x 25	———	16†	2000	Text
02	80 x 25	———	16†	4000	Text
05	80 x 25	640 x 200	2	16000	Graphics
06	40 x 25	320 x 200	4	16000	Graphics
07	80 x 25	640 x 200	4	32000	Graphics
08	40 x 25	320 x 200	16	32000	Graphics

00* Specifies the Processes' current screen.

FF Currently displayed screen. Use when putting several windows on the same physical screen when setting them up from a procedure file. Applications should use $00 instead.

† You have to reconfigure the palette to get 16 colors rather than the default of eight colors. The following section provides information on the palette.

Format Codes 01 and 02 select text screens, and Format Codes 5-8 select graphics screens. The Screen Size column shows the maximum number of text columns and rows available for each screen. The Resolution column shows the maximum *pixels* (graphic units) available for each of the graphic screens. The Memory column shows how much memory NitrOS-9 must set aside for each screen format. Memory requirements depend on the resolution and number of colors selected for a window.

The Palette

BASIC09 has 64 colors you can select for screen displays. The colors are available through a palette. The Color Computer's palette can hold 16 colors at once.

The following chart shows the default colors for the palette in Screen Format 7:

Table 9.6

Register	Color	Register	Color
00	Black	**08**	Black
01	Red	**09**	Green
02	Green	**10**	Black
03	Yellow	**11**	Buff
04	Blue	**12**	Black
05	Magenta	**13**	Green
06	Cyan	**14**	Black
07	White	**15**	Orange

Instead of the default colors, you can select any of the 64 colors (0-63) for any of the palette registers. You do this using the PALETTE command described later in this chapter. The BORDER and COLOR commands also affect the colors available in the palette by changing the color in the background and foreground registers, Registers 02 and 03, respectively.

> **Note:** The information in the next section assumes you have a Color Computer 3 with 512 kilobytes of memory. If your computer has 128 kilobytes of memory, skip to the section "High-Level Graphics With 128K."

Establishing a Graphics Window

To create any window, you should first initialize it with the INIZ command. Type:

iniz w1

If you are not running NitrOS9 EOU, you will need to merge at least stdfonts to any window in order to see text. To do so, type:

```
merge sys/stdfonts>/w1
```

Using the information in the preceding tables, use WCREATE to establish a graphics window. The following command line creates a graphics window in Window 1 that has 320 x 200 resolution and that fills the entire screen. The new window has 16 colors available and provides 40 column by 25 line text:

```
wcreate /w1 -s=8 00 00 40 25 03 02 02
  |      |    |   |  |  |  |  |  |  |
  |      |    |   |  |  |  |  |  |  +-> The screen border color
  |      |    |   |  |  |  |  |  +----> The screen background color
  |      |    |   |  |  |  |  +-------> The screen foreground color
  |      |    |   |  |  |  +----------> The screen height in rows
  |      |    |   |  |  +-------------> The screen width in columns
  |      |    |   |  +----------------> The starting Y-Coordinate
  |      |    |   +-------------------> The starting X-coordinate
  |      |    +-----------------------> The screen type
  |      +----------------------------> The window name
  +-----------------------------------> The command name
```

Starting a Shell in a Window

At this point, the new window exists, and you can send data to it. However, if you want to operate from the window, you must install a shell in it. Type:

```
shell 1=/w1&
```

Press `[CLEAR]` to move to the new window. To load BASIC09, type:

```
basic09 #18K
```

Select either more or less memory, according to your needs. Using BASIC09 in a graphics window, you can write procedures to create high-resolution graphics, and you can display the graphics on the same screen.

Using High-Level Graphics With 128K

If your computer is equipped with only 128 kilobytes of memory, you cannot use more than one window with BASIC09. Also, to use even one window, you must follow certain steps to provide enough memory for BASIC09 operations.

Refer to Table 9.6. You must select a window mode that does not use more than 16000 bytes of memory—either window Format 5 or Format 6.

To provide enough memory to use BASIC09, you must fork a shell to the window you create, then kill the shell in TERM. Doing this means that you can no longer operate from your TERM screen. However, you can run NitrOS-9 and BASIC09 from the window.

The following steps show you how to create a Format 6 graphics screen in Window 1, write a BASIC09 high-resolution graphics procedure, and execute it using minimum memory.

1. Boot NitrOS-9. Then, create a graphics window by typing:
```
iniz w1[ENTER]
wcreate /w1 -s=06 00 00 40 24 06 01 01[ENTER]
merge sys/stdfonts>/w1[ENTER]
shell i=/w1&[ENTER]
ex[ENTER]
```
2. The system stops, and you can no longer type or issue commands. Press `[CLEAR]` to move to the new window. Then, load BASIC09 by typing:
```
basic09[ENTER]
```
3. Enter the edit mode, and type the following procedure:
```
PROCEDURE squeeze
DIM XCOR,YCOR,X,Y:INTEGER; RESPONSE:STRING[1]
RUN GFX2("CUROFF")
XCOR=320 \ YCOR=95 \ X=300 \ FLAG=1
PRINT CHR$(12)
LOOP
FOR Y=1 TO 100 STEP 2
X=X-3
GOSUB 10
IF FLAG<1 THEN
RUN GFX2("COLOR",0)
ELSE
RUN GFX2("COLOR",3)
ENDIF
RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
FLAG=FLAG*-1
NEXT Y
RUN GFX2("COLOR",0)
FOR Y=99 TO 1 STEP -2
```

```
GOSUB 10
X=X+3
RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
NEXT Y
RUN GFX2("COLOR",0)
ENDLOOP
10 RUN INKEY(RESPONSE)
IF RESPONSE="" THEN
RETURN
ENDIF
PRINT CHR$(12)
RUN GFX2("COLOR",9)
RUN GFX2("CURON")
END
```

4. When you have entered the procedure exactly as shown, exit the edit mode, and from the BASICO9 command mode, save Squeeze by typing:
   ```
   save squeeze[ENTER]
   ```
5. Compile Squeeze by typing:
   ```
   pack squeeze[ENTER]
   ```
   Squeeze is now an executable module saved in your current execution directory. The following steps assume your execution directory is /D0/CMDS.
6. Exit BASIC09 by typing:
   ```
   bye[ENTER]
   ```
7. Merge Squeeze, RUNB, INKEY, and GFX2 into one module. To do this, type:
   ```
   merge /d0/cmds/squeeze /d0/cmds/runb
   /d0/cmds/inkey/d0/cmds/gfx2 >
   /d0/cmds/yawn[ENTER]
   ```
8. MERGE does not set the new file Yawn as an executable file. Before you execute it, you must make the file executable by typing:
   ```
   attr /d0/yawn e pe[ENTER]
   ```
9. To execute Yawn, type:\
   ```
   yawn
   ```
10. To terminate the procedure, press the space bar.

The merging procedure in step 7 saves a considerable amount of memory. Every module you load uses one or more 8-kilobyte blocks of storage space. For instance, INKEY is only 94 bytes in length, However, if you load it as a separate module, it requires 8192 bytes. RUNB is 12185 bytes in length. This means that it requires two 8-kilobyte blocks, or 16384 bytes of memory. GFX2 is 2190 bytes in length, and Squeeze is 605 bytes in length. Loaded individually, they also require two memory blocks.

If you load all four modules independently, they use 40960 bytes. However, by combining them into one file, they load into two memory blocks, or 16384 bytes.

Using the information in this section, you can write and execute numerous BASIC09 procedures with only 128 kilobytes of memory. However, if your computer has 512 kilobytes of memory, you can bypass many of these steps. Also, the additional memory enables you to have several windows open at one time. For instance, you can create one window in which to write BASIC09 procedures, another window in which to execute your procedures, and a third window from which you can use NitrOS-9 commands.

> **Note:** The remainder of this chapter assumes you have at least 512 kilobytes of memory. If you don't, you can still run many of the sample procedures by implementing the steps in this section.

Creating Windows from BASIC09

Using GFX2 routines, BASIC09 provides the means to create and manage windows. The steps for creating windows from BASIC09 are as follows:

1. DIM a variable to hold the path number to the window you want to create.
2. OPEN a path to the window.
3. SELECT the new window as the display window.
4. Send commands, data, or text to the window through the open path.
5. CLOSE the open path.
6. Use SELECT to return to your original window.

If you do not want to return immediately to the screen or window of origin, you can skip Steps 5 and 6.

The following sample procedure shows how to open the next available window as a 320 x 192 graphics window, draw a circle, then return to the original screen when you press a key.

```
PROCEDURE make_win
DIM PATH:INTEGER
DIM RESPONSE:STRING[1]
OPEN #PATH,"/W":WRITE
RUN GFX2(PATH,"DWSET",08,00,00,40,24,03,02,02)
RUN GFX2(PATH,"SELECT")
RUN GFX2(PATH,"CIRCLE",200,90,80)
GET #1,RESPONSE
CLOSE #PATH
RUN GFX2("SELECT")
END
```

This procedure establishes a Format 8 window, beginning at coordinates 0,0 and covering the total screen. The foreground color is green, the background color is black, and the border color is black.

Because this procedure does not INIZ the window it opens, the window automatically disappears when the procedure closes its path. To create a window that stays in the system, even after you close the path to it, use INIZ before the OPEN statement, like this:

```
SHELL "INIZ /W2"
```

After you create and define the window, view it by pressing `[CLEAR]`. To get back to the screen you are working on, press `[SHIFT][CLEAR]`. If you intend to use a window more than once in a procedure, you do not need to close its path until the procedure no longer needs it.

Creating Overlay Windows

When you establish a window, you are initializing an NitrOS-9 device. However, an overlay window is only a new screen for an existing window. An overlay screen can be the same size as its window, or it can be smaller. NitrOS-9 automatically transfers to the overlay window any current procedures operating in the device window.

The process for creating overlay windows lets you select whether you want to save the contents of the screen covered by the new window. If you choose to save the contents, the previous screen is redisplayed when you end the overlay.

The following procedure provides an example of using overlay windows. It creates six overlays, each smaller than the preceding window. The procedure then waits for you to press a key. When you do, it removes the overlay windows.

```
PROCEDURE overwindows
DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
DIM RESPONSE:STRING[1]
X=0 \Y=0
X1=80 \Y1=24
PLACE=33
FOR T=1 TO 6
IF T=2 OR T=6 THEN
B=3
ELSE B=2
ENDIF
RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
X=X+6 \Y=Y+2
X1=X1-12 \Y1=Y1-4
FOR J=1 TO 5
PRINT TAB(PLACE); "Overlay Screen "; T
NEXT J
PLACE=PLACE-6
NEXT T
PRINT "Overlay Screen 6"
PRINT "Press A Key...";
GET #1,RESPONSE
FOR T=1 TO 6
RUN GFX2("OWEND")
NEXT T
END
```

The Graphics Cursor and the Draw Pointer

High-resolution graphics provide a text cursor, a graphics cursor, and a *draw pointer*. The text cursor and the graphics cursor can be either visible or invisible. The draw pointer is always invisible.

Text functions always begin at the current location of the text cursor. Whenever you *print* on the screen, the cursor automatically moves to the end of the text or to the beginning of the next line, depending on whether or not you use a semicolon after the print statement. You can reset the text cursor to any place on the screen with the CURXY function of GFX2.

Many BASIC09 graphics functions also begin operating at a location pointed to by the draw pointer. When you begin graphics, the draw pointer is located at coordinates 0,0. BASIC09 then updates the pointer as you execute certain graphics functions. For instance, the LINE function of GFX2 draws from the draw pointer position to the specified end coordinates. The draw pointer is left pointing to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the SETDPTR function to move the draw pointer to new locations.

The graphics cursor is for use with joystick or mouse operations. It provides a *pointer* for graphics applications. The system diskette provides patterns that can be loaded into the graphics cursor *buffer*. You can select from a variety of pointer images.

High-Resolution Text

When you create a graphics window, you can display either text characters, graphics characters, or both.

To display graphics, move the draw pointer to the location where you want the graphics to begin. Then, execute the graphics routines.

To display text, move the text cursor to the location where you want the text to begin. Then, use normal BASIC commands to *print* text.

Instructions for the draw pointer relate to a 640 x 200 grid, numbered 0-639 and 0-199. Instructions for the text cursor relate to the number of characters per line and the number of lines on the current screen format.

Using Fonts

NitrOS-9 Level Two includes fonts (character sets) that are stored in the SYS directory on the system diskette and must be manually merged before you can use them. NitrOS-9 EOU includes many fonts that are pre-loaded and are available for use at startup. You can also create your own fonts and instruct BASIC09 to use them. If you create your own fonts, you can design any symbols or graphics characters you want to use.

To use fonts, you must be in a graphics window. See "Establishing a Graphics Screen" earlier in this chapter. Use the FONT function to tell NitrOS-9 what font you want. The fonts are installed in group 200. The following procedure uses characters installed in group 200, buffers 1, 2, and 3, using the font in buffer 3 to draw a border, then prints a message using the characters in buffer 2. It then returns to buffer 3 and asks you to press a key to end the procedure.

```
PROCEDURE borders
DIM T,B,V,J,K:INTEGER
DIM RESPONSE:STRING[1]
B=199
PRINT CHR$(12)
RUN GFX2("FONT",200,3)
RUN GFX2("COLOR",1,2)
FOR T=0 TO 79
PRINT CHR$(B);
NEXT T
FOR T=1 TO 21
RUN GFX2("CURXY",0,T)
PRINT CHR$(B); CHR$(B);
RUN GFX2("CURXY",78,T)
PRINT CHR$(B); CHR$(B);
NEXT T
RUN GFX2("CURXY",0,21)
FOR T=0 TO 79
PRINT CHR$(B);
NEXT T
RUN GFX2("FONT",200,2)
RUN GFX2("COLOR",0,2)
RUN GFX2("CURXY",45,9)
PRINT "A Demonstration"
RUN GFX2("CURXY",50,10)
PRINT "Of A"
RUN GFX2("CURXY",43,11)
PRINT "Buffer Three Border"
RUN GFX2("CURXY",51,12)
PRINT "And"
RUN GFX2("CURXY",45,13)
PRINT "Buffer Two Text"
RUN GFX2("FONT",200,1)
```

```
RUN GFX2("COLOR",3,2)
RUN GFX2("CURXY",33,15)
PRINT "Press A Key...";
GET #1,RESPONSE
PRINT CHR$(12)
END
```

High-Resolution Quick Reference

High-resolution functions are all part of the GFX2 module. You call them in a BASIC09 procedure with the following syntax:

```
RUN GFX2([PATH,]"FUNCTION"[,PARAMETER[,...]])
```

Path is an optional variable name that tells NitrOS-9 the window in which you want the function performed. *Function* is the high-resolution task you want to perform. *Parameter* is an essential or optional value that affects the performance of the function. Different functions require or permit different numbers of parameters.

The following reference gives a brief description of the high-resolution graphics functions, followed by a detailed description of each function.

Window Functions

Function	Description
CWArea	Changes the size of a window. You can only reduce the working area of a window, not increase it.
DWEnd	Deallocates an established window.
DWProtectSw	Lets you unprotect a window and set other device windows over it. This might destroy the contents of either or both windows.
DWSet	Establishes a window and sets its location on the screen, its size, its background color, its foreground color, and its border color.
GetSel	Returns menu selection
Menu	Set window menus for high level menu handler
Item	Set pulldown items for high level menu handler
OWEnd	Deallocates the specified overlay window.
OWSet	Establishes an overlay window on a device window that already exists. The function also sets the overlay window size, background color, foreground color, and border color. When using this function, you can choose whether or not to save the contents of the original screen.
SBar	Update scroll bars
Select	Selects the window to display.
Title	Set window title & sizes for high level menu handler
UMBar	Update menu bar
WInfo	Returns window information
WnSet	Set high level window type (this is different than basic windows types which set resolution and color depth; these define window types like framed, outlined, etc.)

CWAREA Change working area

Syntax: RUN GFX2([path,]"CWAREA",xcor,ycor,sizex,sizey)

Function: Restricts output in the window to the specified area. The new area must be the same or smaller than the previous working area. When a window's working area is changed, NitrOS-9 scales graphic and text coordinates and graphic images to the new proportions *if the Scaling switch is turned on*. Text characters remain the same size. The xcor,ycor,sizex and sizey parameters are always based on 8x8 pixel fonts, even if a 6x8 pixel font is currently selected.

Parameters:

path	The route to the window in which you want to change the working area.
xcor,ycor	The beginning coordinates (the upper left corner) for the new working area, relative to the original window. The coordinates are based on the character column and row size of the original window.
sizex	Designates the number of columns in the new working area.
sizey	The number of lines available in the new working area.

Examples:

```
RUN GFX2("CWAREA",10,0,40,10)
```

Sample Procedure:

This procedure makes the working area in a window progressively smaller, filling each area with a different color. It then changes the areas' colors rapidly to produce a hypnotic effect.

```
PROCEDURE hypnobox
DIM X,Y,X1,Y1,T,R,COLOR:INTEGER
DIM KEY:STRING[1]
KEY=""
X=3 \Y=1
X1=80-(X+X) \Y1=24-(Y+Y)
FOR T=0 TO 10
RUN GFX2("COLOR",3,T)
RUN GFX2("CLEAR")
RUN GFX2("CWAREA",X,Y,X1,Y1)
X=X+3 \Y=Y+1
X1=80-(X+X) \Y1=24-(Y+Y)
NEXT T
RUN GFX2("COLOR",3,2)
WHILE KEY="" DO
RUN INKEY(KEY)
```

```
FOR T=1 TO 16
R=RND(65)
RUN GFX2("PALETTE",T,R)
NEXT T
ENDWHILE
RUN GFX2("DEFCOL")
RUN GFX2("CWAREA",5,2,80,24)
END
```

DWEND Device window end

Syntax: RUN GFX2([path,]"DWEND")

Function: Deallocates the device window you initialized with DWSET and INIZ. If the window deallocated is the last device window on the screen, BASIC09 returns the screen memory to the system. DWEND automatically positions you in the next device window, a result similar to pressing [CLEAR]. You can use this function with DWSET to redefine a device window to a different type.

Parameters:

path The path number of the window you wish to end. Path can be a constant or variable.

Examples:

```
RUN GFX2("DWEND")
RUN GFX2(PATH,"DWEND")
RUN GFX2(3,"DWEND")
```

Sample Procedure:

From /TERM, this procedure temporarily opens a path to Window 3, displays the new window, draws a design, then returns to the /TERM screen and closes the path.

```
PROCEDURE decorate
DIM PATH,T,Y:INTEGER
OPEN #PATH,"/W3":WRITE
RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
RUN GFX2(PATH,"SELECT")
Y=1
RUN GFX2(PATH,"COLOR",3,2)
FOR T=1 TO 185 STEP 3
Y=Y+1
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
NEXT T
RUN GFX2(PATH,"COLOR",1,2)
FOR T=185 TO 1 STEP -6
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
IF INT(T/3)=T/3 THEN
Y=Y+1
ENDIF
NEXT T
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"DWEND")
CLOSE #PATH
END
```

DWPROTSW Device window protect switch

Syntax: RUN GFX2([path,]"DWPROTSW","switch")

Function: Lets you *unprotect* one device window and set other device windows on top of it.

NitrOS-9 on the Color Computer 3 normally uses a protected windowing system that does not allow window devices to overlap. Removing the window protection with DWPROTSW lets one device window exist on the same screen area as another window device. Because this might destroy the contents of an unprotected window, you need to use care with this function.

As an example, GSHELL uses this to allow you to open other device windows on top of the main window, such as when you select the Clock app from the Tandy menu.

The "bottom device window" that GShell creates covers the entire screen and stays unprotected. Any sizable applications the user picks (like Clock) have their sizing/placement box put on this underlying screen, moved around and sized by the mouse, and once the position and size of the window is locked in, it creates a new device window in that space on top of the unprotected one. More applications can be placed on this same screen, with the position and size of each one shown by drawing on the unprotected window underneath them all.

Parameters:

path	The route to the window you want to unprotect.
switch	Either OFF to turn off protection, or ON to turn on protection. The default is ON.

Examples:

```
RUN GFX2("DWPROTSW",OFF)
RUN GFX2("DWPROTSW",ON)
```

DWSET Device window set

Syntax: RUN GFX2([path,)"DWSET",format,xcor,ycor,width,height, foreground, background, border)

Function: Defines a device window. Normally, you first open a path to a window, then use DWSET to set the window format, location, size, and colors.

Parameters:

path	The route to the window you are defining.
format	The code for the type of screen you want to establish. See Table 9.5 at the beginning of this section for the formats available.
xcor,ycor	The coordinates (character column and row) of the upper left corner of the screen you want to create.
width	The width (in characters) of the new window.
height	The height (in lines) of the new window.
foreground	The code for the window's foreground color.
background	The code for the window's background color.
border	The code for the window's border color.

Examples:

```
RUN GFX2("DWSET",06,50,100,50,10,20,12,9)
```

Sample Procedure:

From /TERM, this procedure temporarily opens a path to the next free window, creates the window format, and uses SELECT to display the new window, draws a design, then returns to the /TERM screen, deallocates the window, and closes the path.

```
PROCEDURE lemon
DIM PATH,T,X,Y:INTEGER
OPEN #PATH,"/W3":WRITE
RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
RUN GFX2(PATH,"SELECT")
Y=1
RUN GFX2(PATH,"COLOR",0,2)
FOR T=1 TO 185 STEP 3
Y=Y+1
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
NEXT T
X=T
RUN GFX2(PATH,"COLOR",3,2)
FOR T=62 TO 1 STEP -3
```

```
RUN GFX2(PATH,"ELLIPSE",326,96,X,T)
IF INT(T/3)=T/3 THEN
X=X+1
ENDIF
NEXT T
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"DWEND")
CLOSE #PATH
END
```

GETSEL Get menu selection or keypress from Multi-Vue menus

Syntax: RUN GFX2([path],"GETSEL",menu_id,menu_item)

Function: Returns the menu number and menu item number (both will be 0 if none was selected) from the Multi-Vue menu handler. If a key was pressed, this can be determined by either an INKEY or an SS.Ready GetStat call. This is usually used in conjunction with ONMOUSE and MOUSE to allow smooth multi-tasking in the background.

For further details and detailed programming examples, see Chapter 13 "Multi-Vue Features".

Parameters:

path	The route to the window that contains the menu.
menu_id	The menu number that was pulled down and clicked on. 0=none selected.
menu_item	The menu item number that was selected with the mouse. 0=none selected.

Examples:

```
RUN GFX2("GETSEL",menunum,menuitem)
```

Sample Procedure:

```
PROCEDURE MemMaps
DIM MId_Memory,menuid,menuitem,Disable, Enable:INTEGER
DIM valid,fire,mx,my,area,sx,sy:INTEGER
DIM wd(2):STRING
DIM m1(4):STRING
DIM progname(4),prompt:STRING[6]
DIM exitflag:BOOLEAN
Disable=0
Enable=1
MId_Memory=33
progname(1)="pmap"
progname(2)="smap"
progname(3)="mmap"
progname(4)="gpmap"
RUN GFX2("DWEnd")
RUN GFX2("DWSet",8,0,0,40,25,3,0,0)
RUN GFX2("curoff")
RUN GFX2("Select")
RUN GFX2("Palette",0,0)
RUN GFX2("Palette",1,7)
RUN GFX2("Palette",2,$38)
RUN GFX2("Palette",3,$3f)
RUN GFX2("Title",wd,"Memory Map Tools",40,20,1)
```

```
RUN GFX2("Menu",wd,1,"Memory Maps",MID_Memory, 6,4,m1,Enable)
RUN GFX2("Item",m1,1,"PMap ",Enable)
RUN GFX2("Item",m1,2,"SMap ",Enable)
RUN GFX2("Item",m1,3,"MMap ",Enable)
RUN GFX2("Item",m1,4,"GPMap",Enable)
RUN GFX2("WnSet",1,wd)
RUN GFX2("SetMouse",3,1,1)
RUN GFX2("GCSet",$ca,1)
exitflag=FALSE
LOOP
RUN GFX2("curoff")
RUN GFX2("OnMouse",0)
RUN GFX2("Mouse",valid,fire,mx,my,area,sx,sy)
IF valid<>0 AND fire=1 AND area=1 THEN
RUN GFX2("GetSel",menuid,menuitem)
IF menuid=MID_Memory AND menuitem>0 AND menuitem<5 THEN
RUN GFX2("OWSet",1,1,1,36,18,0,3)
RUN GFX2("WnSet",4)
RUN GFX2("Font",$c8,2)
SHELL progname(menuitem)
RUN GFX2("Font",$c8,1)
RUN GFX2("curon")
INPUT "Press ENTER:",prompt
RUN GFX2("OWend")
ELSE
IF menuid=2 THEN
exitflag=TRUE
ENDIF
ENDIF
ENDIF
EXITIF exitflag=TRUE THEN
ENDEXIT
ENDLOOP
RUN GFX2("curon")
END
```

MENU Set window menus for high level menu handler

Syntax: RUN GFX2([path],"MENU",d_array,menunum, menuname,menuid,xsize,numofitems,menu_array,enableflag)

Function: Defines the Menus that will appear on the Menu bar. Part of the MultiVue menu handler.

Parameters:

path	The route to the window that contains the menu.
d_array	The window descriptor array.
menunum	Position of the drop down menu in the menu bar (1 to number of menus on menu bar).
menuname	Text name of the menu as it will appear on the menu bar.
menuid	Menu ID number that will be returned to caller (Which menu was dropped down and selected from). 1-32 are reserved to be common ones between programs, some of which are defined in the MultiVue manual. It is recommended that non-general menu id's start at 33 (up to a maximum of 127).
xsize	Width of the pull down menu (in characters, not pixels).
numofitems	Number of items in this pull down menu.
menu_array	Array for this pull down menu.
enableflag	0=Entire pull down menu is disabled (not selectable), 1=pull down menu enabled (selectable).

Examples:

```
RUN GFX2("MENU",wd,1,"Disk",33,8,4,m1,1)
RUN GFX2("MENU",wd,2,"Memory",34,5,2,m2,1)
RUN GFX2("MENU",wd,3,"Special",35,6,3,m2,0)
```

Sample Procedure:
See Sample Program for GETSEL.

ITEM Set pulldown items for high level menu handler

Syntax: RUN GFX2([path,]"ITEM",Menubar_array,itemnum, itemname,enableflag)

Function: Defines a pulldown menu Item. Part of the MultiVue menu handler.

Parameters:

path	The route to the window that contains the menu.
Menubar_array	The menu bar menu selection array. There will be a different array for pull down menu. (See Chapter on MultiVue menus.)
itemnum	Position of this item in the pulldown (1 to number of items on menu).
itenname	Name of this item (text on menu pulldown).
enableflag	0=item is disabled (not selectable), 1=item is enabled (selectable).

Examples:

```
RUN GFX2("ITEM",m1,1,"OPEN",1)
RUN GFX2("ITEM",m1,2,"OPEN",1)
RUN GFX2("ITEM",m1,3,"CLOSE",0)
```

Sample Procedure:
See Sample Program for GETSEL.

OWEND Overlay window end

Syntax: RUN GFX2([path,]"OWEND")

Function: Closes an overlay window. The underlying device window settings are preserved and changes done on the overlay window do not affect those original settings. If the overlay window was created with save switch on (see **OWSET**), whatever was under the overlay window will be restored to the screen.

Parameters:

path path is the path number of the overlay window you wish to end. Path can be a constant or variable.

Examples:

```
RUN GFX2("OWEND")
RUN GFX2(PATH,"OWEND")
RUN GFX2(3,"OWEND")
```

OWSET Establish an overlay window

Syntax: RUN GFX2([path,]"OWSET",save switch,xpos,ypos,xsize, ysize,foreground,background)

Function: Creates an overlay window on a previously existing device window. Reconfigures the current device window paths to use a new area of the screen as the current device window. The underlying window settings are preserved and changes to the overlay do not affect it.

Parameters:

path The route to the window in which you want to set an overlay.
save switch Either 0 or 1. A value of 0 tells BASIC09 not to save the overlaid area. A value of 1 tells BASIC09 to save the overlaid area and restore it when the new window closes.
xpos The character column in which to start the new window (upper left corner)
ypos The character row in which to start the new window (upper left corner).
xsize The width of the new window in characters.
ysize The height of the new window in rows.
foreground The foreground color of the new window.
background The background color of the new window.

Examples:

```
RUN GFX2("OWSET",00,44,10,32,8,00,06)
```

Sample Procedure:

This procedure creates six progressively smaller overlay windows, labeling each. It then waits for you to press a key, after which it erases all the windows and leaves the original window intact.

```
PROCEDURE overwin
DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
DIM RESPONSE:STRING[1]
X=0 \Y=0
X1=80 \Y1=24
PLACE=33
FOR T=1 TO 6
IF T=2 OR T=6 THEN
B=3
ELSE B=2
ENDIF
RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
```

```
X=X+6 \Y=Y+2
X1=X1-12 \Y1=Y1-4
FOR J=1 TO 5
PRINT TAB(PLACE); "Overlay Screen "; T
NEXT J
PLACE=PLACE-6
NEXT T
PRINT "Press A Key...";
GET #1,RESPONSE
FOR T=1 TO 6
RUN GFX2("OWEND")
NEXT T
END
```

SBAR Update position of scrollbars in a framed with scrollbars window

Syntax: RUN GFX2([path,]"SBAR",xcol,yrow)

Function: Moves the horizontal and vertical scroll bars to the specified positions. These are base 0, and the maximum positions depends on the size of the window. The coordinates are based on text character positions (8x8 pixels).

Parameters:

xcol X position of horizontal scroll bar (in columns). 0 to (window width-4).
yrow Y position of vertical scroll bar (in rows). 0 to (window height-5).

Examples:

```
RUN GFX2("SBAR",0,10)
```

Sample Procedure:
Run 'FontInfo'. You will need #16K for running within BASIC09.

```
PROCEDURE FontInfo
DIM errnum,MId_Font,menuid,menuitem,Disable, Enable:INTEGER
DIM ybarpos,linenum,startline,endline,
numoffonts,valid,fire,mx,my,area,sx,sy:INTEGER
DIM wd(2),m1(1):STRING
DIM filepath:BYTE
DIM header,fonttext(100):STRING[100]
DIM prompt:STRING[6]
DIM exitflag:BOOLEAN
Disable=0
Enable=1
MId_Font=33
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
RUN GFX2("curoff")
RUN GFX2("Select")
PRINT "Setting palettes from GSHPAL settings in
/dd/sys/env.file..."
RUN GetGSHPal
RUN GFX2("Title",wd,"Font List",80,25,1)
RUN GFX2("Menu",wd,1,"Font Info",MID_Font, 6,1,m1,Disable)
RUN GFX2("Item",m1,1,"Info",Enable)
RUN GFX2("WnSet",2,wd)
RUN GFX2("SetMouse",3,1,1)
RUN GFX2("GCSet",$ca,4)
PRINT "Getting font list from /dd/sys/fontlist.txt..."
numoffonts=0
ON ERROR GOTO 10
OPEN #filepath,"/dd/sys/fontlist.txt":read
linenum=1
```

```
REPEAT
READ #filepath,fonttext(linenum)
linenum=linenum+1
UNTIL LEFT$(fonttext(linenum-1),4)="----"
header=fonttext(linenum-2)
RUN GFX2("CLEAR")
WHILE NOT(EOF(#filepath)) DO
numoffonts=numoffonts+1
EXITIF numoffonts>100 THEN
numoffonts=100
ENDEXIT
READ #filepath,fonttext(numoffonts)
ENDWHILE
CLOSE #filepath
ON ERROR
startline=1
exitflag=FALSE
endline=startline+21
IF endline>numoffonts THEN
endline=numoffonts
ENDIF
RUN GFX2("GCSet",0,0)
RUN GFX2("curxy",0,0)
RUN GFX2("revon")
PRINT header;
RUN GFX2("ereoline")
RUN GFX2("revoff")
RUN GFX2("CWArea",1,2,78,22)
FOR linenum=startline TO endline
RUN GFX2("curxy",0,linenum-1)
PRINT fonttext(linenum);
RUN GFX2("ereoline")
NEXT linenum
RUN GFX2("GCSet",$ca,1)
LOOP
RUN GFX2("curoff")
RUN GFX2("OnMouse",0)
RUN GFX2("Mouse",valid,fire,mx,my, area,sx,sy)
IF valid<>0 AND fire=1 AND area=1 THEN
RUN GFX2("GetSel",menuid,menuitem)
(* Scroll up
IF menuid=4 THEN
(* Check if already top (do nothing if we are)
IF startline>1 THEN
startline=startline-1
endline=endline-1
IF endline<numoffonts THEN
endline=endline+1
ENDIF
RUN GFX2("curxy",0,0)
RUN GFX2("INSLIN")
PRINT fonttext(startline);
RUN CalcBarPos(21+0,numoffonts, startline,ybarpos)
RUN GFX2("SBar",0,ybarpos)
```

```
ENDIF
ELSE
(* Scroll down
IF menuid=5 THEN
(* Check if already bottom (do nothing if we are)
IF endline+1<=numoffonts THEN
endline=endline+1
startline=startline+1
RUN GFX2("curxy",0,0)
RUN GFX2("DELLIN")
RUN GFX2("curxy",0,21)
PRINT fonttext(endline);
RUN CalcBarPos(21+0,numoffonts, startline,ybarpos)
RUN GFX2("SBar",0,ybarpos)
ENDIF
ELSE
(* Close box
IF menuid=2 THEN
exitflag=TRUE
ENDIF
ENDIF
ENDIF
ENDIF
EXITIF exitflag=TRUE THEN
ENDEXIT
ENDLOOP
RUN GFX2("CurOn")
GOTO 20
10 errnum=ERR
RUN GFX2("OWSet",1,10,10,60,7,0,3)
RUN GFX2("WnSet",4)
PRINT "Could not read /dd/sys/fontlist.txt"
PRINT "Error #";ERR
SHELL "ERROR "+STR$(errnum)
RUN GFX2("curon")
INPUT "Press <ENTER> to exit:",prompt
RUN GFX2("OWEnd")
20 RUN GFX2("WnSet",0)
RUN GFX2("curon")
END

PROCEDURE GetGshPal
(* Get GSHPAL settings, set palettes 0-3
(* If any errors, just return
DIM envfile:BYTE
DIM envline:STRING[128]
DIM tempstr:STRING[32]
DIM setpal(4):BYTE
DIM baseshift,colorvalue(3):INTEGER
DIM c,palclr,tempnum:INTEGER
setpal(1)=$1b
setpal(2)=$31
setpal(3)=0
```

```
setpal(4)=0
ON ERROR GOTO 5
OPEN #envfile,"/dd/sys/env.file":READ
WHILE (NOT(EOF(#envfile))) DO
READ #envfile,envline
IF LEFT$(envline,1)<>"*" THEN
RUN ForceUpper(envline)
IF LEFT$(envline,6)="GSHPAL" THEN
setpal(3)=VAL(MID$(envline,7,1))
IF setpal(3)>=0 AND setpal(3)<=3 THEN
tempstr=RIGHT$(envline,LEN(envline)-SUBSTR("=",envline))
palclr=0
FOR c=0 TO 2
colorvalue(c+1)=VAL(MID$(tempstr,c*2+1,1))
NEXT c
FOR c=3 TO 1 STEP -1
baseshift=VAL(MID$("0189",colorvalue(c)+1,1))
palclr=palclr+baseshift
IF c=2 THEN
palclr=palclr+baseshift
ELSE
IF c=1 THEN
palclr=palclr+baseshift*3
ENDIF
ENDIF
NEXT c
setpal(4)=palclr
PUT #1,setpal
ENDIF
ENDIF
ENDIF
ENDWHILE
CLOSE #envfile
5 ON ERROR
END

PROCEDURE ForceUpper
DIM curpos:INTEGER
PARAM str(128):BYTE
FOR curpos=1 TO 128
EXITIF str(curpos)=$FF THEN
ENDEXIT
IF str(curpos)>=$61 AND str(curpos)<=$7A THEN
str(curpos)=str(curpos)-$20
ENDIF
NEXT curpos
END

PROCEDURE CalcBarPos
(* Entry:
(*   barsize=# of row or columns in the scroll bar
(*   numofelements=# of elements (lines or columns) in file we
are viewing
(*   elementnum=starting element # to view
```

```
(*   (all of the above are base 1)
(* Exit:
(*   barposition=position to use in SBar (base 0)
PARAM barsize,numofelements, elementnum, barposition:INTEGER
DIM startelement,ydiff,ypos:REAL
(* Adjust start line # to base 0
startelement=elementnum-1
(* If # of elements completely fits on screen, barposition
always 0
IF numofelements<=barsize THEN
barposition=0
ELSE
IF startelement+barsize>=numofelements THEN
barposition=barsize-1
ELSE
ydiff=FLOAT(barsize)/(numofelements-barsize)
IF startelement=1 THEN
barposition=0
ELSE
barposition=INT(ydiff*startelement)
ENDIF
ENDIF
ENDIF
END
```

SELECT Select window to view on screen

Syntax: RUN GFX2([path,]"SELECT")

Function: SELECT causes a window to display if the procedure is operating in the active window. If the procedure is not in the active window, the newly selected window displays when you press [CLEAR]. If you do not specify a path, BASIC09 selects the device using the standard input, standard output, and standard error paths, Paths 0, 1, and 2.

Parameters:

path The path to the window to select.

Examples:

```
RUN GFX2("SELECT")
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"SELECT")
```

Sample Procedure:

From /TERM, this procedure temporarily opens a path to the next free window, creates the window format, and uses SELECT to display the new window, draws a design, then returns to the /TERM screen, deallocates the window, and closes the path.

```
PROCEDURE design
DIM PATH,T,Y:INTEGER
OPEN #PATH,"/W":WRITE
RUN GFX2(PATH,"DWSET",5,0,0,80,24,3,2,2)
RUN GFX2(path,"PALETTE",1,9)
RUN GFX2(path,"CUROFF")
RUN GFX2(PATH,"SELECT")
Y=1
FOR T=1 TO 200 STEP 3
Y=Y+1
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
NEXT T
RUN GFX2(PATH,"COLOR",1,2)
FOR T=200 TO 1 STEP -6
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
NEXT T
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"DWEND")
CLOSE #PATH
END
```

TITLE Sets up the main Multi-Vue style window descriptor

Syntax: RUN GFX2([path,],"TITLE",d_array,title$,xcolmin,ycolmin, menucount)

Function: Sets up the main window descriptor for Multi-Vue style windows, as well as the window title (shown when not the active window), the minimum size the window can be run in (by text columns/rows, which are 8x8 pixel chunks), and the maximum number of user-defined menu bar drop down menus. Note that the Close box is always present and does not need to be defined by the program and is not included in this count.

Parameters:

path	The route to the window that you are setting up a Multi-Vue style window.
d_array	The window descriptor array. It should be STRING[32] and the array should be the maximum number of menu pull downs you want +1 (i.e. 2 menu pulldowns should be DIM wd(3):STRING).
title$	The title of the window. This will only display if the window is not the current active window (has control of the keyboard and mouse). Maximum 20 characters.
xcolmin	The minimum size (in text columns) that the window can run in.
ycolmin	The minimum size (in text rows) that the window can run in.
menucount	The number of menu pull downs that will appear on the menu bar.

Examples:

```
RUN GFX2("TITLE",wd,"Tools",34,10,2)
```

Sample Procedure:
See the sample program for GETSEL.

UMBAR **Updates the menu bar, allowing enabling/disabling of menu items or entire menus under program control. It also allows a program to do its own pulldowns.**

Syntax: RUN GFX2([path,],"UMBAR")

Function: Updates the Menu bar and Menu pulldowns to change any activated/deactivated that the program updated through the menu and item arrays. If one makes a menu with no items, they can create their own dropdown using OWSet, and enable/disable the menu bar menu entry when needed. MVCanvas uses this for its Tools menu (which is all graphics and no text), for example.

Parameters:

path The route to the window that you are updating the menu bar on.

Examples:

```
RUN GFX2("UMBAR")
```

Sample Procedure:

```
PROCEDURE FileUtils
DIM MId_File,MId_Utils,menuid,menuitem, Disable,Enable:INTEGER
DIM valid,fire,mx,my,area,sx,sy:INTEGER
DIM wd(3):STRING
DIM m1(4):STRING
DIM m2(4):STRING
DIM filename:STRING
DIM progname(3),prompt:STRING[6]
DIM openedflag,exitflag:BOOLEAN
Disable=0
Enable=1
MId_File=33
MId_Utils=34
progname(1)="FStat"
progname(2)="List"
progname(3)="Dump"
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
RUN GFX2("curoff")
RUN GFX2("Select")
RUN GFX2("Palette",0,0)
RUN GFX2("Palette",1,7)
RUN GFX2("Palette",2,$38)
RUN GFX2("Palette",3,$3f)
RUN GFX2("Title",wd,"File Utilities",80,25,2)
RUN GFX2("Menu",wd,1,"File",MID_File,6,2, m1,Enable)
RUN GFX2("Item",m1,1,"Open",Enable)
RUN GFX2("Item",m1,2,"Close",Enable)
RUN GFX2("Menu",wd,2,"Utilities",MID_Utils,6,3, m2,Disable)
```

```
RUN GFX2("Item",m2,1,"FStat ",Enable)
RUN GFX2("Item",m2,2,"List ",Enable)
RUN GFX2("Item",m2,3,"Dump ",Enable)
RUN GFX2("WnSet",1,wd)
RUN GFX2("SetMouse",3,1,1)
RUN GFX2("GCSet",$ca,1)
exitflag=FALSE
openedflag=FALSE
LOOP
RUN GFX2("curoff")
RUN GFX2("OnMouse",0)
RUN GFX2("Mouse",valid,fire,mx,my,area,sx,sy)
IF valid<>0 AND fire=1 AND area=1 THEN
RUN GFX2("GetSel",menuid,menuitem)
IF menuid=2 THEN
exitflag=TRUE
ELSE
IF menuid=MID_File THEN
IF menuitem=2 AND openedflag=TRUE THEN
openedflag=FALSE
RUN GFX2("Menu",wd,2,"Utilities",MID_Utils,6,3, m2,Disable)
RUN GFX2("UMBAR")
PRINT CHR$(12);
ELSE
IF menuitem=1 THEN
SHELL "DIR"
RUN GFX2("CURON")
INPUT "Type in filename to open:",filename
RUN GFX2("CUROFF")
IF TRIM$(filename)<>"" THEN
openedflag=TRUE
RUN GFX2("Menu",wd,2,"Utilities",MID_Utils,6,3, m2,Enable)
RUN GFX2("UMBAR")
PRINT CHR$(12);"'";filename;"' selected"
ELSE
PRINT CHR$(12)
ENDIF
ENDIF
ENDIF
ELSE
IF menuid=MID_Utils AND menuitem>=1 AND menuitem<=3 THEN
RUN GFX2("OWSet",1,2,2,76,21,0,3)
RUN GFX2("WnSet",4,wd)
SHELL progname(menuitem)+" "+filename
RUN GFX2("CURON")
INPUT "Press ENTER:",prompt
RUN GFX2("CUROFF")
RUN GFX2("OWend")
ENDIF
ENDIF
ENDIF
ENDIF
EXITIF exitflag=TRUE THEN
```

```
ENDEXIT
ENDLOOP
RUN GFX2("curon")
END
```

WINFO Returns window information about the current window

Syntax: RUN GFX2([path,]"WINFO",format,xsize,ysize,foreground, background,border)

Function: Returns the screen type, window size and current foreground, background and border colors for the process's window (or window on the optional path). Very useful for writing programs that adapt to the size of the window that they are run on.

Parameters:

format	Screen type. See Table 9.5 at the beginning of this section.
xsize	Current working area window width.
ysize	Current working area window height.
foreground	Foreground color.
background	Background color.
border	Border color.

Examples:

```
RUN GFX2("WINFO",sctype,curwidth,curheight,
fgcolor,bkcolor,brdcolor)
```

Sample Procedure:

```
PROCEDURE winfo
DIM sctype,curwidth,curheight,fgcolor, bkcolor,brdcolor:INTEGER
DIM c,orgwidth,orgheight:INTEGER
DIM prompt:STRING[2]
GOSUB 100
orgwidth=curwidth
orgheight=curheight
RUN GFX2("OWSet",1,5,5,30,15,2,3)
PRINT "After overlay window,"
GOSUB 100
RUN GFX2("OWEnd")
RUN GFX2("CWArea",0,10,40,15)
FOR c=1 to 20
PRINT
NEXT c
PRINT "After CWArea,"
GOSUB 100
RUN GFX2("CWArea",0,0,orgwidth,orgheight)
RUN GFX2("CurXY",0,orgheight-1)
END
100 RUN GFX2("WINFO",sctype,curwidth,curheight,
fgcolor,bkcolor,brdcolor)
PRINT "Screen type=";sctype
```

```
PRINT "Current width=";curwidth
PRINT "Current height=";curheight
PRINT "Foreground color=";fgcolor
PRINT "Background color=";bkcolor
PRINT "Foreground color=";brdcolor
INPUT "Press [ENTER]:",prompt
RETURN
```

WNSET Actually sets the Multu-Vue window type (both with menus and not)

Syntax: RUN GFX2([path,],"WNSET",wintype,d_array)

Function: Actually draws the Multi-Vue style window. NOTE: You must have set up your TITLE, MENU, and ITEM functions first (and the main window array) if you are going to be using any type of framed window (with or without scroll bars). You can also use WNSET on overlay windows as well. The window that WNSET is going to be changing the appearance of must be already set up via DWSET or OWSET as WNSET simply changes the existing window's appears for the Multi-Vue window type that is desired.

Parameters:

path The route to the window that you are setting up a Multi-Vue style window.

wintype The window frame type, from the table below. Only 1 and 2 allow pull down menus. Note that only type 0 will let you print or draw in the entire window; window types >0 reserve the far edge columns and rows for borders, menus, etc.:

1. = normal window
2. = framed menu window
3. = framed menu window with scroll bars
4. = shadowed box window
5. = double box window
6. = plain box window

d_array The window descriptor array. It should be STRING[32] and the array should be the maximum number of menu pull downs you want +1 (i.e. 2 menu pulldowns should be DIM wd(3):STRING)

Examples:

```
RUN GFX2("WnSet",1,wd)
```

Sample Procedure:

See the sample program for GETSEL.

Drawing Functions

Function	Description
Arc	Draws an arc
Bar	Draws a filled rectangle
Box	Draws a rectangle outline
Circle	Draws a circle
Draw	Draws an image from directions provided in a draw string
Ellipse	Draws an ellipse
FCircle	Draws a filled Circle
FEllipse	Draws a filled ellipse
Fill	Fills the area of the window the same color as the pixel under the draw pointer
Line	Draws a line
Point	Sets the pixel under the draw pointer to the specified color or to the default color

ARC Draw an arc

Syntax: RUN GFX2({path,)"ARC"[,mx,my],xrad,yrad,xcor1,ycor1,xcor2, ycor2)

Function: Draws an arc at the current or specified draw position with the specified X and Y radius. If you specify the same radius for both X and Y, the function draws a circular arc, otherwise the arc is elliptical. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199.

ARC begins drawing from the point on the screen closest to the first set of coordinates (xcor1, ycor1). It stops at the portion of the screen closest to the second set of coordinates (xcor2, ycor2). You can determine on which side of the line ARC draws by selecting which set of coordinates is the beginning and which set is the end.

Parameters:

path	The route to the window in which you want to draw an arc.
mx,my	The X- and Y-coordinates for the center of the arc. If you do not specify mx and my, BASIC09 uses the current draw pointer position.
xrad	The radius of the arc's width.
yrad	The radius of the arc's height.
xcor1 ycor1 *xcor2,ycor2*	The beginning and ending coordinates for an imaginary line from which the function draws an arc. The line is relative to the center of the arc (the center point is at 0,0 for these coordinates) and extends through the two coordinates from one edge of the screen to the other. Negative coordinates are allowed.

Examples:

```
RUN GFX2("ARC",58,108,50,108,50,156)
```

Sample Procedure:
This procedure draws a series of diagonally-cut arcs on a graphics window screen.

```
PROCEDURE arcing
DIM MX,MY,XRAD,YRAD,XCOR,YCOR,XCOR2,YCOR2:INTEGER
DIM T,X,Y,Z:INTEGER
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
PRINT CHR$(12)
FOR T=1 TO 98 STEP 2
RUN GFX2("ARC",318,95,150,T,0,1,0,1)
```

```
RUN GFX2("ARC",324,95,150,T,1,0,1,1)
NEXT T
```

BAR Fill a rectangle

Syntax: RUN GFX2([path,]"BAR"[,xcor1,ycor1],xcor2,ycor2)

Function: Fills a rectangular area defined by two sets of coordinates. BAR defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199. BAR resets the draw pointer to its original position.

Parameters:

path The route to the window in which you want to draw a bar.

xcor1,ycor1 The beginning coordinates of the line defining the area to fill. If you omit these coordinates, BAR uses the draw pointer position.

See the previous section "The Graphics Cursor and The Draw Pointer." Also see SETDPTR.

xcor2,ycor2 The ending coordinates of the line defining the area to fill.

Examples:

```
RUN GFX2("BAR",200,100)
RUN GFX2("BAR",0,0,100,50)
```

Sample Procedure:

This procedure draws a bar chart on a window screen.

```
PROCEDURE OSgraf
DIM COLOR,T,X,XCOR1,YCOR1,XCOR2,YCOR2:INTEGER;
RESPONSE:STRING[1]
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
PRINT CHR$(12)
RUN GFX2("DEFCOL")
COLOR=13 \ XCOR1=10 \ YCOR1=180
XCOR2=-XCOR1+40
RUN GFX2("CUROFF")
FOR T=1 TO 10
READ YCOR2
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",XCOR1,YCOR1,XCOR2,YCOR2)
RUN GFX2("COLOR",7)
RUN GFX2("BOX",XCOR1,YCOR1,XCOR2,YCOR2)
COLOR=COLOR+1 \ XCOR1=XCOR1+50 \ XCOR2=XCOR1+40
NEXT T
PRINT \ PRINT "    NitrOS-9 Sales Chart"
RUN GFX2("BOX",0,0,510,180)
GET #1,RESPONSE
```

```
RUN GFX2("CURON")
PRINT CHR$(12)
END
DATA 170,150,140,130,110,90,70,60,50,30
```

BOX Draw a rectangle

Syntax: RUN GFX2([path,]"BOX"[,xcor1,ycor1],xcor2,ycor2)

Function: Draws a rectangle. BOX defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199. BOX resets the draw pointer to its original position.

Parameters:

path	The route to the window in which you want to draw a box.
xcor1,ycor1	The beginning coordinates for the line that defines the rectangle to be drawn. If you omit these coordinates, BOX uses the draw pointer position.
xcor2,ycor2	The ending coordinates for the line that defines the rectangular area to be drawn.

Examples:

```
RUN GFX2("BOX",200,100)
RUN GFX2("BOX",0,0,100,50)
```

Sample Procedure

This procedure draws a series of progressively smaller boxes of different colors on a window screen. Then, it rapidly changes the colors of the boxes to produce a hypnotic effect.

```
PROCEDURE hypbox
DIM X,Y,X1,Y1,T,R,COLOR: INTEGER
DIM KEY:STRING[1]
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
KEY=""
X=18 \Y=6
Y1=185 \X1=621
RUN GFX2("CLEAR")
FOR T=0 TO 15
COLOR=T
RUN GFX2("COLOR",3)
RUN GFX2("BOX",X,Y,X1,Y1)
RUN GFX2("COLOR",COLOR)
RUN GFX2("FILL",X-1,Y-1)
X=X+18 \Y=Y+6
X1=X1-18 \Y1=Y1-6
NEXT T
WHILE KEY="" DO
RUN INKEY(KEY)
FOR T=1 TO 16
```

```
  R=RND(65)
  RUN GFX2("PALETTE",T,R)
  NEXT T
  ENDWHILE
  RUN GFX2("DEFCOL")
  END
```

CIRCLE Draw a circle

Syntax: RUN GFX2([path,]"CIRCLE"[,xcor,ycor],radius)

Function: Draws a circle with a specified radius. If you specify coordinates, CIRCLE uses them for the center point. Otherwise, CIRCLE locates the center of the circle at the current draw pointer position. See "The Graphics Cursor and the Draw Pointer" earlier in this section. Also see SETDPTR.

Parameters:

path	The route to the window in which you want to draw a circle.
xcor,ycor	The coordinates for the circle's center. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199.
radius	The radius of the circle.

Examples:

```
RUN GFX2("CIRCLE",100)
RUN GFX2("CIRCLE",100,200,50)
```

Sample Procedure:
This procedure uses circles to produce a geometric design.

```
PROCEDURE ciraround
DIM T,X,Y:INTEGER
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
PRINT CHR$(12)
RUN GFX2("COLOR",1,2)
FOR T=1 TO 130
X=150*SIN(T)+320
Y=25*C0S(T)+96
RUN GFX2("CIRCLE",X,Y,100)
NEXT T
RUN GFX2("COLOR",3,2)
FOR T=1 TD 45
X=150*SIN(T)+320
Y=25+C0S(T)+96
RUN GFX2("CIRCLE",X,Y,100)
NEXT T
END
```

DRAW Draw a polyline figure

Syntax: RUN GFX2([path,]"DRAW",[xcor,ycor],option list)

Function: Draws, in the directions specified, and for the distances specified, in an option list. It allows an optional start location before the option list. The option list is a string of characters and numbers. You can separate options with spaces or commas or run them together. You must include commas between the two coordinates for the B and U options and the optional start position.

Parameters:

path	The route to the window in which you want to draw.
xcor,ycor	Optional start coordinates. Sets the draw pointer to the specified coordinates before drawing.
option list	A string consisting of one or more of the following options:

Options:

N*num*	draws north (up) num units.
S*num*	draws south (down) num units.
E*num*	draws east (right) num units.
W*num*	draws west (left) num units.
NE*num*	draws northeast (up and right) num units.
NW*num*	draws northwest (up and left) num units.
SE*num*	draws southeast (down and right) num units.
SW*num*	draws southwest (down and left) num units.
A*val*	rotates the draw axis. Possible values are:

1. = normal
2. = 90 degrees
3. = 180 degrees
4. = 270 degrees

U*xcor,ycor*	draws a relative vector to the specified coordinates. Xcor and ycor are relative to the current draw pointer position. The draw pointer location does not change. Xcor and ycor must be separated by a comma.
B*xcor,ycor*	produces a blank line (moves the cursor but does not draw). The xcor and ycor coordinates are relative to the current draw pointer location. If you specify relative coordinates located offscreen, you cannot see subsequent lines.

Examples:

```
RUN GFX2("DRAW","N10,E10,S10,W10")
RUN GFX2("DRAW",155,95,"N10E10S10W10")
```

Sample Procedure:

```
PROCEDURE drawing
DIM T,X,Y,COLOR:INTEGER
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
COLOR=0
RUN GFX2("CLEAR")
FOR T=1 TO 96 STEP 6
RUN GFX2("SETDPTR",320,96)
FOR Y=0 TO 3
COLOR=MOD(Y,2)
RUN GFX2("COLOR",COLOR)
FOR X=1 TO 4
READ DR$
DR$="A"+STR$(Y)+DR$ +STR$(T)
RUN GFX2("DRAW",DR$)
NEXT X
NEXT Y
RESTORE
NEXT T
RUN GFX2("COLOR",3)
END
DATA "N","E","S","W"
```

ELLIPSE Draw an ellipse

Syntax: RUN GFX2([path,)"ELLIPSE"[,xcor,ycor],xrad,yrad)

Function: Draws an ellipse with the center at the current draw pointer position or at the specified X,Y coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199.

Parameters:

path	The route to the window in which you want to draw.
xcor,ycor	The coordinates for the ellipse's center. If you omit these coordinates, ELLIPSE uses the current draw pointer position.
xrad,yrad	The radii of the ellipse's width and height.

Examples:

```
RUN GFX2("ELLIPSE",100,50)
RUN GFX2C"ELLIPSE",100,125,100,10)
```

Sample Procedure:
This program uses ELLIPSE to draw a graphic design shaped like a Christmas tree decoration.

```
PROCEDURE xbulb
DIM T,Y:INTEGER
RUN GFX2("DWEnd")
RUN GFX2("DWSet",7,0,0,80,25,3,0,0)
Y=1
RUN GFX2("COLOR",3,2)
RUN GFX2("CLEAR")
FOR T=1 TO 180 STEP 3
Y=Y+1
RUN GFX2("ELLIPSE",320,96,T,Y)
NEXT T
RUN GFX2("COLOR",1,2)
FOR T=180 TO 1 STEP -6
RUN GFX2("ELLIPSE",320,96,T,Y)
IF INT(T/3)=T/3 THEN
Y=Y+1
ENDIF
NEXT T
RUN GFX2("COLOR",3,2)
END
```

FCIRCLE Draw a filled (painted) circle

Syntax: RUN GFX2([path,]"FCIRCLE"[,xcor,ycor],radius)

Function: Draws a filled in circle with a specified radius. If you specify coordinates, FCIRCLE uses them for the center point. Otherwise, FCIRCLE locates the center of the circle at the current draw pointer position. See "The Graphics Cursor and the Draw Pointer" earlier in this section. Also see SETDPTR.

Parameters:

Path	The route to the window in which you want to draw a circle.
xcor,ycor	The coordinates of the circle's center. The X coordinates are in the range of 0-639. The Y coordinates are in the range of 0-199.
Radius	The radius of the circle. This value *must* be an INTEGER type.

Examples:

```
RUN GFX2("FCIRCLE",100)
RUN GFX2("FCIRCLE",100,200,50)
```

Sample Procedure:
This program generates a bulls-eye target.

```
PROCEDURE Bullseye
DIM C:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
RUN GFX2("CUROFF")
FOR C=0 TO 7
RUN GFX2("COLOR",LAND(C,3))
RUN GFX2("FCIRCLE",320,100,160-C*20)
NEXT C
RUN GFX2("CURON")
END
```

FELLIPSE Draw a filled (painted) ellipse

Syntax: RUN GFX2([path],"FELLIPSE"[,xcor,ycor],xrad,yrad)

Function: Draws a filled in ellipse with center at the current draw pointer position or at the specified X,Y coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199.

Parameters:

path	The route to the window in which you want to draw a circle.
xcor,ycor	The coordinates of the ellipse's center. If you omit these coordinates, ELLIPSE uses the current draw pointer position.
xrad,yrad must	The radii of the ellipse's width and height. These values
	be INTEGER types.

Examples:

```
RUN GFX2("FELLIPSE",100,50)
RUN GFX2("FELLIPSE",100,50,200,50)
```

Sample Procedure:
This program generates an elliptical bulls-eye target.

```
PROCEDURE BullseyeEllipse
DIM C:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
RUN GFX2("CUROFF")
FOR C=0 TO 7
RUN GFX2("COLOR",LAND(C,3))
RUN GFX2("FELLIPSE",320,100,320-C*20,100-C*10)
NEXT C
RUN GFX2("CURON")
END
```

FILL Fill (paint) window

Syntax: RUN GFX2([path,]"FILL"[,xcor,ycor])

Function: Paints an area with the current foreground color. Paint fills the portion of the window that is the same color as the pixel under the draw pointer.

NOTE: FILL can error out if it has a very complex area to fill. If this happens, try to use FILL in simpler areas multiple times rather than one large complex fill.

Parameters:

path	The route to the window in which you want to use the FILL function.
xcor,ycor	Are optional X- and Y-coordinates to reposition the draw pointer before FILL begins. If you omit these coordinates, BASIC09 uses the current draw position.

Examples:

```
RUN GFX2("FILL",100,180)
```

Sample Procedure:
This procedure draws and fills 100 boxes on a window.

```
PROCEDURE colorbox
DIM A,B,C,D,T,COLOR:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
COLOR=0
RUN GFX2("CLEAR")
FOR T=1 TO 100
A=RND(560)
B=RND(151)
C=A+RND(80)
D=B+RND(40)
COLOR=COLOR+1
RUN GFX2("COLOR",COLOR)
RUN GFX2("BOX",A,B,C,D)
RUN GFX2("FILL",A+1,B+1)
NEXT T
```

LINE Draw a line

Syntax: RUN GFX2([path,]"LINE"[,xcor1,ycor1],xcor2,ycor2)

Function: Draws a line in one of the following ways:

- From the current draw pointer to the specified X- and Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X- and Y-coordinates.
- **NOTE:** Horizontal lines are the fastest, followed by vertical lines, followed by diagonal lines.

Parameters:

path	The route to the window in which you want to draw a line.
xcor1,ycor1	The optional beginning X- and Y-coordinates for the line.
xcor2,ycor2	The ending X- and Y-coordinates for the line.

Examples:

```
RUN GFX2("LINE",192,128)
RUN GFX2("LINE",0,0,192,128)
```

Sample Procedure:

This procedure draws a sine wave of vertical lines across a window.

```
PROCEDURE waves
DIM A,X,Y,Z:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
CALC=0
A=100
RUN GFX2("CLEAR")
RUN GFX2("COLOR",3,2)
FOR X=0 TO 638 STEP 1
CALC=CALC+.05
Y=A-SIN(CALC)*15
Z=Y+25
RUN GFX2("LINE",X,Y,X,Z)
NEXT X
END
```

POINT Mark a point

Syntax: RUN GFX2([path,]"POINT"[,xcor,ycor])

Function: Sets the pixel at the current draw pointer position or at the specified coordinates to the current foreground color. If you do not specify coordinates, POINT sets the pixel at the draw pointer.

Parameters:

path	The route to the window in which you want to turn on the specified pixels.
xcor,ycor	Optional coordinates for the POINT function. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-199.

Examples:

```
RUN GFX2("POINT")
RUN GFX2("POINT",192,128)
```

Sample Procedure:
This procedure uses POINT to produce a swirl design on a window screen.

```
PROCEDURE paint
BASE 0
DIM X(20),Y(20):INTEGER
DIM T,R,J,K:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
J=0
K=0
RUN GFX2("CUROFF")
RUN GFX2("CLEAR")
FOR T=1 TO 288 STEP 3
J=J+1
FOR R=0 TO 11
X(R)=INT(T*SIN(30+R+K))+320
Y(R)=INT(J*COS(30+R+K))+96
RUN GFX2("POINT",X(R),Y(R))
K=K+1
NEXT R
NEXT T
RUN GFX2("CURON")
END
```

Configuring Functions

Function	Description
Border	Sets the border palette register
Color	Sets any of the foreground, background, or border colors
DefCol	Sets palette registers to the default colors
GCSet	Establishes a buffer from which BASIC09 gets the graphics cursor
Logic	Turns on AND, OR, or XOR logic functions for draw functions
Palette	Changes colors in the palette registers
Pattern	Establishes a buffer from which BASIC09 gets a pattern for graphics functions
PutGC	Positions the graphics cursor
ScaleSw	Turns scaling on or off
SetDPtr	Positions the draw pointer

BORDER Set the border color

Syntax: RUN GFX2([path,]"BORDER",color)

Function: Resets the palette register that affects a window's border color (register 0) to the specified color code. For information on the palette and on screen colors, see "The Palette" and Table 9.5 earlier in this chapter.

Parameters:

path The route to the window in which you want to change border color.

color One of the current palette colors. *Color* can be either a constant or a variable.

Examples:

```
RUN GFX2("BORDER",1)
```

Sample Procedure:
This procedure lets you select different border colors by pressing [+] or [-] to select higher or lower color codes. Press [Q] to end the procedure.

```
PROCEDURE border
DIM COLOR:INTEGER
DIM KEY:STRING[1]
COLOR=8
RUN GFX2("CLEAR")
WHILE KEY<>"q" AND KEY<>"Q" DO
GET #1,KEY
IF KEY="-" OR KEY="=" THEN
COLOR=COLOR-1
ENDIF
IF KEY="+" OR KEY=";" THEN
COLOR=COLOR+1
ENDIF
IF COLOR>8 OR COLOR<0 THEN COLOR=8
ENDIF
RUN GFX2("BORDER",COLOR)
RUN GFX2("CURXY",0,0)
ENDWHILE
END
```

COLOR Set screen colors

Syntax: RUN GFX2([path,]"COLOR",foreground[,background] [,border])

Function: Changes any of the foreground, background, or the border colors. COLOR does not change the draw pointer position.

Parameters:

path	The route to the window in which you want to change one or more screen or text colors.
foreground	The register number for the foreground palette.
background	The register number for the background palette.
border	The register number for the border palette. Changing the border color for any window on a screen, changes the border color for all windows on the same screen.

Examples:

```
RUN GFX2("COLOR",1)
RUN GFX2("COLOR",1,2)
RUN GFX2("COLOR",1,2,1)
```

Sample Procedure:

This procedure fills a window screen with multicolored circles.

```
PROCEDURE bubbles
DIM X,Y,W,Z,T:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
Z=1
RUN GFX2("COLOR",1,0,0)
RUN GFX2("CLEAR")
FOR T=1 TO 80
X=RND(635)+4
Y=RND(185)+5
W=RND(50+5)
Z=Z+1
IF Z>3 THEN Z=1
ENDIF
RUN GFX2("CIRCLE",X,Y,W)
RUN GFX2("COLOR",Z)
RUN GFX2("FILL",X,Y)
NEXT T
RUN GFX2("COLOR",3,2,2)
END
```

DEFCOL Set default colors

Syntax: RUN GFX2([path,]"DEFCOL")

Function: Sets the palette registers back to their default values. The type of monitor you have determines the actual hues. See "The Palette" and Table 9.6 earlier in this section.

Parameters:

path The route to the window in which you want to restore the original palette registers.

Examples:

```
RUN GFX2("DEFCOL")
```

GCSET Set graphics cursor

Syntax: RUN GFX2("GCSET",group,buffer)

Function: Defines a buffer from which BASIC09 gets the graphics cursor. This lets you define your own cursor for graphics operations. To turn the graphics cursor off, use a group Number 0. You must execute this command to display a graphics cursor. Before using GCSET, you must merge the Stdcur file in the SYS directory to the window.

NOTE: The larger the graphics cursor is, the slower it will draw/redraw on the screen. Also, for optimal speed, a graphics cursor should be the same color depth as the screen it is running on. (a 2 color buffer for a 2 color screen, a 4 color buffer for a 4 color screen, a 16 color buffer for a 16 color screen). Please note that the standard graphics cursors in OS9/NitrOS9 are *only* 2 color, so they will run a bit slower on a 4 or 16 color screen.

Parameters:

group	The group number of the buffer containing the cursor image to use. See NitrOS-9 Windowing System for information on the group to use.
buffer	The number of the buffer that contains the cursor image to use. See NitrOS-9 Windowing System for information on the buffer to use.

Examples:

```
RUN GFX2("GCSET",1,5)
```

LOGIC Perform logic function

Syntax: RUN GFX2("LOGIC","function")

Function: Causes BASIC09 to perform the specified logic function on all data bits used by subsequent drawing functions. Once set, the logic function remains in effect until you turn LOGIC off.

Parameters:

function can be any of the following logical functions:

OFF	—	no logic
AND	—	performs AND logic
OR	—	performs OR logic
XOR	—	performs XOR logic

Examples:

```
RUN GFX2("LOGIC","AND")
RUN GFX2("LOGIC","XOR")
```

Sample Procedure:

This procedure uses LOGIC to draw a horizontal bar across a background of multicolored vertical bars. Using XOR logic, the procedure causes the horizontal bar to change the color of each vertical bar.

```
PROCEDURE logic
DIM A,Z,T,X,Y,COLOR:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
RUN GFX2("LOGIC","OFF")
RUN GFX2("CLEAR")
COLOR=0
FOR T=0 TO 619 STEP 20
COLOR=COLOR+1
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",T,0,T+20,190)
NEXT T
RUN GFX2("COLOR",3,2)
RUN GFX2("LOGIC","XOR")
FOR T=1 TO 10
RUN GFX2("BAR",0,80,639,112)
NEXT T
RUN GFX2("LOGIC","OFF")
END
```

PALETTE Set color for palette registers

Syntax: RUN GFX2([path,]"PALETTE",register,color)

Function: Sets palette colors. Lets you install any of the Color Computer's 64 colors in the palette for use with text and graphics.

Parameters:

path	The route to the window where you want to change palette colors.
register	The number of the register in which you want to install a new color.
color	The code of the new color you want to install.

Examples:

```
RUN GFX2("PALETTE",13,32)
```

Sample Procedure:
This procedure draws a series of bars and circles, then repeatedly changes their colors using PALETTE.

```
PROCEDURE palette
DIM T,K,J,X,Y,COLOR:INTEGER
DIM RESPONSE:STRING[1]
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
RUN GFX2("COLOR",3,2,2)
COLOR=0
RUN GFX2("CLEAR")
RUN GFX2("CUROFF")
FOR Y=0 TO 23 STEP 3
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",0,Y,639,Y+3)
COLOR=COLOR+1
IF COLOR=2 THEN
COLOR=COLOR+1
ENDIF
NEXT Y
FOR Y=164 TO 185 STEP 3
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",0,Y,639,Y+3)
COLOR=COLOR+1
NEXT Y
COLOR=0
FOR K=45 TO 170 STEP 48
FOR T=100 TO 580 STEP 100
RUN GFX2("COLOR",3)
RUN GFX2("CIRCLE",T,K,30)
RUN GFX2("COLOR",COLOR)
```

```
RUN GFX2("FILL",T,K)
COLOR=COLOR+1
IF COLOR=2 THEN
COLOR=COLOR+1
ENDIF
NEXT T
NEXT K
REPEAT
X=RND(63)
REPEAT
Y=RND(16)+1
UNTIL Y<>2
RUN GFX2("PALETTE",Y,X)
RUN INKEY(RESPONSE)
UNTIL RESPONSE>""
RUN GFX2("DEFCOL")
RUN GFX2("CURON")
END
```

PATTERN Select pattern buffer

Syntax: RUN GFX2([path,]"PATTERN",group,buffer)

Function: Selects the contents of a preloaded Get/Put buffer as a pattern for graphics functions. Although PATTERN can use a buffer of any size, it uses a specific number of bytes, depending on the screen format in use:

Color Mode	Pattern Array Size	Bits Per Pixel
02	4 bytes x 8 bytes = 32 bytes	1
04	8 bytes x 8 bytes = 64 bytes	2
16	16 bytes x 8 bytes = 128 bytes	4

The pattern array is a 32 x 8 pixel representation of graphics memory. It takes the current color mode into consideration to define the number of bits per pixel and pixels per byte. If the buffer is larger than the number of bytes required, PATTERN ignores the extra bytes. BASIC09 uses the selected pattern with all draw commands until you change the pattern or turn off the pattern function by specifying a group and buffer number of 0.

NOTE: One should use patterns for the Color Mode that matches the screen type that they are using to get the maximum speed. If they don't match, the graphics driver has to do a pixel by pixel conversion for every drawing command that is using the pattern, which is much slower. NitrOS9 EOU preloads the standard and MVCanvas pattern buffers for all 3 color depths (2, 4 and 16 color).

Parameters:

path The route to the window in which you want to use a new graphics pattern,

group The group number of the buffer you want to use for a graphics pattern.

buffer The buffer number that you want to use for a graphics pattern.

Examples:

```
RUN GFX2("PATTERN",1,3)
```

Sample Procedure:

This procedure loads the current window data at location 0,0 into a buffer to use as a draw pattern. It then draws a circle and fills the circle with the pattern in the buffer.

```
PROCEDURE pattern
TYPE registers=cc,a,b,dp:BYTE; x,y,u:INTEGER
DIM regs:registers
DIM buffernum:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
buffernum=-1
RUN GFX2("DWEND")
RUN GFX2("DWSET",8,0,0,40,25,3,2,2)
WHILE buffernum<>0 DO
RUN GFX2("COLOR",4)
RUN GFX2("CLEAR")
RUN GFX2("FCIRCLE",320,96,100)
RUN GFX2("CURXY",0,23)
RUN GFX2("ERLINE")
INPUT "Buffer #0-16, 0=quit:",buffernum
EXITIF buffernum=0 THEN
ENDEXIT
RUN GFX2("PATTERN",205,buffernum)
RUN GFX2("COLOR",3)
RUN GFX2("FILL",320,96)
RUN GFX2("PATTERN",0,0)
regs.x=60
RUN SysCall($0A,regs)
ENDWHILE
END
```

PUTGC Put graphics cursor

Syntax: RUN GFX2([path,]"PUTGC",xcor,ycor)

Function: Places and displays the graphics cursor at the specified location. Use screen relative coordinates for this function, not window relative coordinates. The horizontal range is 0-639. The vertical range is 0-199.

Parameters:

path	The route to the window where you want to display a graphics cursor.
xcor,ycor	The screen coordinates for the cursor location. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-199.

Examples:

```
RUN GFX2("PUTGC",100,5)
```

Sample Procedure:
This procedure displays the available graphic cursors stored in group 202. Before this procedure can work, you must merge the Stdptrs file in the SYS directory of your system disk with the window you are using. For instance, if your system diskette is in Drive /D0, merge Stdptrs with Window 1, by typing:

```
merge /d0/sys/stdptrs > /w1
```

NitrOS9/EOU pre-loads the standard graphic cursors.

```
PROCEDURE viewcur
DIM T,Z:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,0,2,2)
RUN GFX2("CLEAR")
FOR T=1 TO 7
RUN GFX2("GCSET",202,T)
RUN GFX2("PUTGC",320,96)
FOR Z=1 TO 6000
NEXT Z
NEXT T
RUN GFX2("GCSET",0,0)
END
```

SCALESW Enable/disable scaling

Syntax: RUN GFX2([path,]"SCALESW","switch")

Function: Enables or disables scaling when drawing on variously formatted windows. Scaling in windows is normally on. If scaling is off, coordinates are relative to the window origin coordinates. Scaling does not affect text.

Parameters:

path The route to the window where you want to turn scaling off or on.
switch Either OFF (disable scaling} or ON (enable scaling).

Examples:

```
RUN GFX2("SCALESW","OFF")
```

Sample Procedure:
This procedure runs a routine of drawing a design in overlay windows twice. The routine runs once with scaling off and once with scaling on. After the first routine pauses, press the space bar to see the second demonstration.

```
PROCEDURE scale
DIM X,Y,X1,Y1,T,B,J,R,W,Z:INTEGER
DIM RESPONSE:STRING[1]
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RUN GFX2("PALETTE",1,9)
RUN GFX2("CUROFF")
RUN GFX2("CLEAR")
PRINT "The first set of windows will be with scaling off."
PRINT "The second set of windows will be with scaling on."
INPUT "Press [ENTER] to begin:",response
FOR J=1 TO 2
IF J=1 THEN
RUN GFX2("SCALESW","OFF")
ELSE
RUN GFX2("SCALESW","ON")
ENDIF
X=0 \Y=0 \X1=80 \Y1-24
FOR T=1 TO 4
IF T=2 OR T=4 THEN
B=3
ELSE
B=2
ENDIF
RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
FOR R=1 TO 35
W=40*SIN(R)+170
Z=25+COS(R)+45
```

```
RUN GFX2("CIRCLE",W,Z,30)
NEXT R
X=X+6 \Y=Y+2 \X1=X1-12 \Y1=Y1-4
NEXT T
PRINT "Press A Key...";
GET #1,RESPONSE
FOR T=1 TO 4
RUN GFX2("OWEND")
NEXT T
NEXT J
END
```

SETDPTR Set draw pointer

Syntax: RUN GFX2([path,]"SETDPTR",xcor,ycor)

Function: Places the draw pointer at the specified coordinates. The draw pointer selects the beginning point of the next graphics draw function (such as CIRCLE, LINE, BOX, and so on), if you do not supply other coordinates.

Parameters:

path	The route to the screen where you want to set the draw pointer.
xcor,ycor	The screen coordinates for the draw pointer location. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0- 199.

Examples:

```
RUN GFX2("SETDPTR",100,5)
```

Sample Procedure:
This procedure uses coordinates from a DATA statement for setting the draw pointer to create a series of star shapes.

```
PROCEDURE star
DIM X,Y,T,J:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
PRINT CHR$(12)
FOR J=1 TO 10
READ X,Y
RUN GFX2("SETDPTR",X+J,Y+J+J)
FOR T=1 TO 5
READ X,Y
RUN GFX2("LINE",X+J,Y+J+J)
NEXT T
NEXT J
DATA 320,46,440,146,200,84,440,84,200,146,320,46
END
```

Get/Put Functions

Function	Description
DefBuff	Defines a buffer for storage
Get	Saves a specified portion of a window to a buffer
GPLoad	Preloads a buffer from a disk file
KillBuff	Deallocates a buffer
Put	Places the image stored in a buffer onto a window

DEFBUFF Define a GET/PUT buffer

Syntax: RUN GFX2("DEFBUFF",group,buffer,size)

Function: Defines a buffer for GET/PUT operations and other purposes.

When you define a buffer, you do so by group number and buffer number. Each buffer you define can allocate up to 48K (free RAM permitting). These buffers can be mapped into a process space to be manipulated (assuming enough free space in the process' RAM along with its program and data areas) that can then be used for GET/PUT graphics, clipboards and other data storage. The system needs 30 bytes overhead (buffer definitions) for GET/PUT buffers. Within the group, you can allocate one or more buffers. Select a group number and a buffer number as indicated in the following "Parameters" section. Use these numbers in future references to the buffer.

A buffer remains allocated until you use the KILLBUFF function to remove it from your system's memory. For more information on GET/PUT buffers, see KILLBUFF, PUT, GET, and GPLOAD.

Parameters:

group	A number you select in the range 1-199.
buffer	A number (in the range 1-255) that you assign to the buffer you create.
size	The size of the buffer, in the range of 1 to 49152 bytes, depending on available memory in your system.

NOTE: If the size is >32767, it will need to be specified in hexadecimal format. Example: To make a buffer of 40000 bytes, pass the size as '$9C40'.

Notes: One method of selecting a group number is to use the ID function to obtain your user's process ID number. Then, use this ID number as the base group number. A general guideline is that system wide buffers, such as fonts, mouse cursors, etc., use reserved groups 200 to 254 (some of those are reserved for future use). Group 255 is not directly accessible to a user process; this group is reserved for overlay windows.

If a program needs more than 255 buffers then their "home" group number can provide, it can use additional group numbers that are their home group number plus multiples of 32, as long as the result is <200. (So process #3 could use groups 3,35,67,99,131,163; this would allow for up to 1,530 buffers for a single process).

Examples:

```
RUN GFX2("DEFBUFF",1,5,$4000)
```

GET Get a block from the window

Syntax: RUN GFX2([path,]"GET",group,buffer,xcor,ycor,xsize,ysize)

Function: Saves a window area Get/Put buffer. Use PUT to replace the image to the window. If you did not previously define the buffer, BASIC09 creates it. If you store the window data in a predefined buffer, the data must be the same size or smaller than the buffer, If not, BASIC09 truncates the data to the size of the buffer. (Also see PUT and DEFBUFF.)

Parameters:

path	The route to the window where you want to save an image.
group	The group number of the Get buffer (1-199).
buffer	The Get buffer number (1-255).
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window image to save. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-199.

Note: You can GET/PUT on a hardware text screen, and the X and Y ranges on those are 0-39 or 0-79 (40 and 80 column text screen respectively, and 0-24 on the Y range. Also, a buffer going off the right side of the screen wraps around to the next line(s) on the left.

xsize	The horizontal size of the window section to save.
ysize	The vertical size of the window section to save.

Examples:

```
RUN GFX2("GET",1,5,0,0,10,15)
```

Sample Procedure:
This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation.

```
PROCEDURE putdown
□DIM T,J:INTEGER
□RUN GFX2("DWEND")
□RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2C"COLOR",3)
□RUN GFX2C"FILL",320,90)
```

```
RUN GFX2("BAR",305,100,335,104)
RUN GFX2("GET",1,1,288,85,50,23)
RUN GFX2("GET",1,2,1,1,50,23)
RUN GFX2("PUT",1,2,288,85)
J=10
FOR T=20 TO 559 STEP 4
J=J+1
RUN GFX2("PUT",1,1,T,J)
NEXT T
RUN GFX2("KILLBUFF",1,1)
RUN GFX2("CURON")
END
```

GPLOAD Load data into Get/Put buffer

Syntax: RUN GFX2("GPLOAD",group,buffer,format,xdim,ydim,size)

Function: Loads a buffer with image data that PUTBLK can use for window displays. If the Get/Put buffer is not created, BASICO9 creates it. If it is defined, the size of the data should not be larger than the buffer. This command allows a Get/Put buffer to be loaded from disk or created from scratch, amongst other things.

Parameters:

group	The group number you select, in the range 1-199, to let you group buffers.
buffer	A number in the range 1-255 that you assign to the buffer you create.
format	The type code of the screen format. (See Table 9-3.)
xdim	The X (horizontal) dimension of the stored block.
ydim	The Y (vertical) dimension of the stored block.
size	The size of the buffer in bytes. A buffer size can be up to 32 kilobytes, depending on available memory.

Examples:

```
RUN GFX2("GPLOAD",1,5,6,100,50,5000)
```

KILLBUFF Deallocate Get/Put buffer

Syntax: RUN GFX2("KILLBUFF",group,buffer)

Function: Deallocates the indicated Get/Put buffer, returning it's memory to the system. You select group and buffer numbers when you define a buffer or when you load or get a window image. For more information on Get/Put buffers, see DEFBUFF, PUT, GET, and GPLOAD.

Parameters:

group	The group number of the buffer you want to deallocate, in the range 1-199. Buffer Group Numbers 0 and 200-255 are reserved for NitrOS-9 system use.
buffer	The number of the buffer to deallocate, in the range 1-255. If you want to kill an entire group of buffers, use buffer 0.

Examples:

```
RUN GFX2("KILLBUFF",1,5)
```

Sample Procedure:
This procedure draws a figure on a window screen, loads it into a buffer, then repeatedly places it in new locations on the screen. Each new PUT erases the previous image.

```
PROCEDURE putdown
DIM X,Y,T,J:INTEGER
RUN GFX2("CUROFF")
RUN GFX2("CLEAR")
RUN GFX2("ELLIPSE",320,96,12,4)
RUN GFX2("CIRCLE",320,90,5)
RUN GFX2("COLOR",1)
RUN GFX2("FILL",320,96)
RUN GFX2("COLOR",3)
RUN GFX2("FILL",320,90)
RUN GFX2("BAR",305,100,335,104)
RUN GFX2("GET",1,1,288,85,50,23)
RUN GFX2("GET",1,2,1,1,50,23)
RUN GFX2("PUT",1,2,288,85)
J=10
FOR T=20 TO 559 STEP 6
J=J+2
RUN GFX2("PUT",1,1,T,J)
NEXT T
RUN GFX2("KILLBUFF",1,1)
RUN GFX2("CURON")
END
```

PUT Put a saved data block on the window

Syntax: RUN GFX2([path,]"PUT",group,buffer,xcor,ycor)

Function: Places the image in the specified Get/Put buffer on the window. PUT requires only the group and buffer numbers and the window coordinates for the upper left corner of the image. The GET function saves the dimensions of the block in the buffer. PUT automatically handles window format conversion; however it should be noted that if conversion is needed (different color depth between the buffer and the window), it greatly slows down. Clipping currently also slows things down, except for clipping on the bottom. Hardware text windows now allow GET/PUT as well, requiring 2 bytes per character, but they will wrap to the next line on the right side rather than clip. It should also be noted that PUT is heavily optimized when drawn on even byte boundaries horizontally that match the original GET (vertical does not matter). This is every 2 pixels in a 16 color mode, every 4 pixels in a 4 color mode, and every 8 pixels in a 2 color mode. The speed gains can be several times faster.

Parameters:

path	The route to the window where you want to place a pre-saved image.
group	The group number of the buffer in which to save the window data. If this your own and not a system one, it is recommended that you use your process ID (see the ID function) as the primary group number. See GET for further details.
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window position. The X-coordinates are in the range of 0-639. The Y-coordinates are in the range 0-199. On hardware text windows, they are 0-39 (40 column) or 0-79 (80 column) for the X-coordinate, and 0-24 for the Y coordinate.

Note: You can GET/PUT on a hardware text screen, and the X and Y ranges on those are 0-39 or 0-79 (40 and 80 column text screen respectively, and 0-24 on the Y range. Also, a buffer going off the right side of the screen wraps around to the next line(s) on the left.

Examples:

```
RUN GFX2("PUT",1,5,100,50)
```

Sample Procedure:

This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation. The program will also demonstrate the speed difference between even byte boundary PUT and non-even byte boundary PUT. Note that the even byte version is actually drawing more frames than the non-even one (90 times vs. 135) yet is still much faster.

```
PROCEDURE putdown
DIM x,y,t,j,idnum:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RUN GFX2("CUROFF")
RUN GFX2("CLEAR")
RUN GFX2("PALETTE",1,9)
RUN GFX2("ID",idnum)
RUN GFX2("ELLIPSE",320,96,12,4)
RUN GFX2("CIRCLE",320,90,5)
RUN GFX2("COLOR",1)
RUN GFX2("FILL",320,96)
RUN GFX2("COLOR",3)
RUN GFX2("FILL",320,90)
RUN GFX2("BAR",305,100,335,104)
RUN GFX2("GET",idnum,1,288,85,50,23)
RUN GFX2("GET",idnum,2,0,1,50,23)
RUN GFX2("PUT",idnum,2,288,85)
RUN GFX2("CURXY",40,1)
PRINT "Non-boundary PUT";
j=10
FOR t=20 TO 559 STEP 6
j=j+2
RUN GFX2("PUT",idnum,1,t,j)
NEXT t
RUN GFX2("CURXY",40,1)
PRINT "Even boundary PUT";
j=10
FOR t=20 TO 559 STEP 4
j=j+1
RUN GFX2("PUT",idnum,1,t,j)
NEXT t
RUN GFX2("KILLBUFF",idnum,0)
RUN GFX2("CURXY",0,23)
RUN GFX2("CURON")
END
```

Text/Cursor Handling Functions

Function	Description
Bell	Sounds the terminal bell
Clear	Homes the cursor and clears the screen
CrRtn	Performs a carriage return by moving the cursor down one line and to the extreme left of the window
CurDwn	Moves the graphics cursor down one line
CurHome	Positions the cursor at coordinates 0,0
CurLft	Moves the graphics cursor left one space
CurOff	Turns the graphics cursor off
CurOn	Turns the graphics cursor on
CurRgt	Moves the graphics cursor right one space
CurUp	Moves the graphics cursor up one line
CurXY	Positions the cursor at specified coordinates
DelLin	Deletes the line at the graphics cursor position
ErEOLine	Erases from the cursor to the end of the line
ErEOWndw	Erases from the graphics cursor to the end of the window
ErLine	Erases the line under the cursor
InsLin	Inserts a blank line at the graphics cursor position

BELL Ring the terminal bell

Syntax: RUN GFX2("BELL")

Function: Rings the terminal's bell (produces a beep through the speaker).

Parameters:

None

Examples:

```
RUN GFX2("BELL")
```

CLEAR Clear the screen

Syntax: RUN GFX2([path,]"CLEAR")

Function: Clears the current working area of a window. CLEAR does not change the location of the draw pointer but does set the text cursor and graphics cursor location to the upper left corner of the window.

Parameters:

path The route to the window you want to clear.

Examples:

```
RUN GFX2("CLEAR")
```

CRRTN Carriage return

Syntax: RUN GFX2([path,]"CRRTN")

Function: Causes BASIC09 to send a carriage return to a window. The cursor moves down one line and to the extreme left of the window.

Parameters:

path The route to the window in which you want a carriage return.

Examples:

```
RUN GFX2("CRRTN")
```

CURDWN Cursor down

Syntax: RUN GFX2([path,]"CURDWN")

Function: Moves the cursor down one text line. The X-coordinate, or column position, remains the same.

Parameters:

path The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURDWN")
```

CURHOME Cursor home

Syntax: RUN GFX2([path,]"CURHOME")

Function: Moves the text cursor to the top left corner of the screen.

Parameters:

path The route to the window where you want to reset the cursor.

Examples:

```
RUN GFX2("CURHOME")
```

CURLFT Move cursor left

Syntax: RUN GFX2(path,]"CURLFT")

Function: Moves the cursor one character to the left.

Parameters:

path The route to the window where you want to move the cursor.

Examples:

```
RUN GFX2("CURLFT")
```

CUROFF Turn off cursor

Syntax RUN GFX2([path,]"CUROFF")

Function: Makes the cursor invisible.

NOTE: Shutting off the text cursor will speed up text output slightly. On graphics windows, it will also speed up graphics commands, as the system will not need to remove and redraw the cursor before and after every graphics command.

Parameters:

path The route to the window in which you want to turn the cursor off.

Examples:

```
RUN GFX2("CUROFF")
```

CURON Turn on cursor

Syntax: RUN GFX2([path,]"CURON")

Function: Makes the text cursor visible.

NOTE: Turning on the text cursor will slow down text output slightly. On graphics windows, it will also slow down graphics commands, as the system will need to remove and redraw the cursor before and after every graphics command.

Parameters:

path The route to the window in which you want to turn the cursor on.

Examples:

```
RUN GFX2("CURON")
```

CURRGT Move cursor right

Syntax: RUN GFX2([path,]"CURRGT")

Function: Moves the cursor one character to the right.

Parameters:

path The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURRGT")
```

CURUP Move cursor up

Syntax: RUN GFX2([path,]"CURUP")

Function: Moves the cursor up one line.

Parameters:

path The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURUP")
```

CURXY Set cursor position

Syntax: RUN GFX2([path,]"CURXY",column,row)

Function: Moves the cursor to the specified column and row position. The column and row coordinates are relative to the window's current character width and height.

Parameters:

path The route to the window in which you want to move the cursor.

column The column (horizontal) position for the cursor. Please note that if you are on a graphics screen and have a 6 pixel wide font selected, that the column will properly take that into consideration. The following table lists the column values.

	Font Width	
Types	**6 Pixel**	**8 Pixel**
6,8	0-53	0-39
5,7	0-105	0-79

row The row (vertical) position for the cursor.

Examples:

```
RUN GFX2("CURXY",10,10)
```

DELLIN Delete current line of text

Syntax: RUN GFX2([path,]"DELLIN")

Function: Deletes the line on which the cursor is resting and closes the space. DELLIN operates on both text and graphics screens.

Parameters:

path The route to the window in which you want to delete a line.

Examples:

```
RUN GFX2("DELLIN")
```

Sample Procedure:
This procedure draws a series of various colored concentric circles, then produces a lemon shape by removing slices of the circle with DELLIN

```
PROCEDURE slice
DIM X,Y,R,T,COLOR:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RUN GFX2("CLEAR")
COLOR=0
X=320
Y=96
FOR T=185 TO 10 STEP -10
RUN GFX2("CIRCLE",X,Y,T)
NEXT T
FOR T=140 TO 320 STEP 10
RUN GFX2("COLOR",COLOR)
RUN GFX2("FILL",T,96)
COLOR=COLOR+1
NEXT T
RUN GFX2("CURXY",0,8)
FOR T=1 TO 8
RUN GFX2("DELLIN")
NEXT T
RUN GFX2("COLOR",3,2)
END
```

EREOLINE Erase to end of line

Syntax: RUN GFX2([path,]"EREOLINE")

Function: Deletes the portion of the current line from the cursor to the right side of the window.

Parameters:

path The route to the window in which you want to erase a portion of a line.

Examples:

```
RUN GFX2("EREOLINE")
```

Sample Procedure:
This procedure uses EREOLINE to produce a series of steps down the screen.

```
PROCEDURE steps
DIM T,J,K:INTEGER
RUN GFX2("COLOR", 2,3)
RUN GFX2("CLEAR")
RUN GFX2("COLOR",3,2)
FOR T=0 TO 22
J=T*3
RUN GFX2("CURXY",J,T)
RUN GFX2("EREOLINE")
NEXT T
```

EREOWNDW Erase to end of window

Syntax: RUN GFX2([path,]"EREOWNDW")

Function: Deletes all the lines in a window from the line on which the cursor is positioned to the bottom of the window.

Parameters:

path The route to the window in which you want to delete screen contents.

Examples:

```
RUN GFX2("EREOWNDW")
```

ERLINE Delete current line of text

Syntax: RUN GFX2([path,]"ERLINE")

Function: Deletes the current line {(on which the cursor is resting) from the window but does not close the space.

Parameters:

path The route to the window in which you want to remove the contents of a screen line.

Examples:

```
RUN GFX2("ERLINE")
```

Sample Procedure:
This procedure draws a bull's-eye design, then slices it with the ERLINE function.

```
PROCEDURE cut
DIM X,Y,R,T,COLOR:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
COLOR=0
X=320
Y=96
RUN GFX2("CLEAR")
COLOR=0
FOR T=185 TO 10 STEP -10
RUN GFX2("CIRCLE",X,Y,T)
NEXT T
FOR T=140 TO 320 STEP 10
RUN GFX2("COLOR",COLOR)
RUN GFX2("FILL",T,96)
COLOR=COLOR+1
NEXT T
FOR T=2 TO 22 STEP 2
RUN GFX2("CURXY",0,T)
RUN GFX2("ERLINE")
NEXT T
RUN GFX2("COLOR",3,2)
END
```

INSLIN Insert line

Syntax: RUN GFX2([path,]"INSLIN")

Function: Moves the window lines at and below the cursor down one line.

Parameters:

path The route to the window in which you want a blank line.

Examples:

```
RUN GFX2("INSLIN")
```

Sample Procedure:
This procedure draws a round face on the screen, then uses INSLIN and DELLIN to make a mouth appear to move.

```
PROCEDURE chomp
DIM X,Y,T:INTEGER
DIM RESPONSE:STRING[1]
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RESPONSE=""
RUN GFX2("CLEAR")
RUN GFX2("CIRCLE",320,96,80)
RUN GFX2("COLOR",0,2)
RUN GFX2("FILL",320,96)
RUN GFX2("COLOR",2)
RUN GFX2("CIRCLE",285,80,12)
RUN GFX2("CIRCLE",355,80,12)
RUN GFX2("FILL",285,80)
RUN GFX2("FILL",355,80)
RUN GFX2("CIRCLE",315,96,3)
RUN GFX2("CIRCLE",325,96,3)
RUN GFX2("ARC",320,92,14,3,3,1,1,1)
RUN GFX2("COLOR",3,2)
RUN GFX2("CIRCLE",289,77,3)
RUN GFX2("CIRCLE",359,77,3)
RUN GFX2("CURXY",0,14)
REPEAT
RUN GFX2("INSLIN")
FOR X=1 TO 100
NEXT X
RUN GFX2("DELLIN")
RUN INKEY(RESPONSE)
UNTIL RESPONSE>""
END
```

Font Handling Functions

Function	Description
BlinkOff	Turns blinking characters off (only for hardware text screens)
BlinkOn	Turns blinking characters on (only for hardware text screens)
BoldSw	Selects or deselects bold characters
Font	Specifies the buffer from which BASIC09 selects its font characters
PropSw	Selects or deselects proportional characters
RevOff	Turns reverse video off
RevOn	Turns reverse video on
TCharSw	Transparent characters switch
UndInOff	Turns the underline function off
UndlnOn	Turns the underline function on

BLNKOFF Character blink off

Syntax: RUN GFX2([path,]"BLNKOFF")

Function: Executing BLNKOFF causes all subsequent characters sent to a window on a hardware screen to stop blinking. A hardware screen is one of the predefined device windows /W1 through /W7. Executing BLNKOFF cancels a previous blink command; characters already blinking continue to do so. Blink does not operate on graphics windows.

Parameters:

path The route to the window in which you want to blink characters.

Examples:

```
RUN GFX2("BLNKOFF")
```

BLNKON Character blink on

Syntax: RUN GFX2([path,]"BLNKON")

Function: Executing BLNKON causes all subsequent characters sent to a window on a hardware screen to blink. A hardware screen is one of the predefined device windows /W1 through /W7. Executing BLNKOFF cancels a previous blink command; characters already blinking continue to do so. Blink does not operate on graphics windows.

Parameters:

path The route to the window in which you want to blink characters.

Examples:

```
RUN GFX2("BLNKON")
```

BOLDSW Switch bold characters on or off

Syntax: RUN GFX2([path,]"BOLDSW","switch")

Function: Causes characters to display in either regular or bold typeface. The default is regular typeface. BOLD only works on graphics screens.

Parameters:

path	The route to the window in which you want bold characters.
switch	Can be either "ON" or "OFF." I switch is "ON," subsequent characters are bold. If switch is "OFF," subsequent characters are not bold.

Examples:

```
RUN GFX2("BOLDSW","ON")
```

Sample Procedure:
This procedure demonstrates the BOLDSW function by displaying both bold and normal text on a window screen.

```
PROCEDURE bold
DIM LINE:STRING
DIM LETTER:STRING[1]
DIM T,J,K,FLAG:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RUN GFX2("CLEAR")
FLAG=1
FOR T=1 TO 8
READ LINE
FOR J=1 TO LEN(LINE)
LETTER=MID$(LINE,J,1)
IF LETTER<>"!" AND LETTER<>"#" THEN
PRINT LETTER;
ENDIF
IF LETTER="1" THEN
FLAG=FLAG*-1
IF FLAG>0 THEN
RUN GFX2("BOLDSW","OFF")
ELSE
RUN GFX2("BOLDSW","ON")
ENDIF
ENDIF
IF LETTER="#" THEN
PRINT CHR$(34);
ENDIF
NEXT J
PRINT
NEXT T
```

```
PRINT \ PRINT
END
DATA "This is a demonstration of"
DATA "the !Bold! function of"
DATA "BASIC09's GFX2 module."
DATA "Use the command"
DATA "!RUN GFX2(#BOLDSW#,#0N#)1"
DATA "to turn boldface on."
DATA "Use !RUN GFX2(#BOLDSW#,#OFF#)!"
DATA "to turn boldface off"
```

FONT Define font buffer

Syntax: RUN GFX2([path,]"FONT",group,buffer)

Function: Defines a buffer from which BASIC09 gets the character font (style) for the current screen. Use the text/cursor handling functions referenced in this section with the font you load. When you merge the Stdfonts file in your SYS directory with a graphics window, you have the choice of three fonts from Buffers 1, 2, and 3, located in Group 200. You can also create your own fonts. FONT works only on graphics screen. See "Using Fonts" earlier in this chapter.

You must load the font you want to use into the defined buffer before using FONT.

NOTE: While the standard for fonts is that all fonts are in group 200 ($C8), it is not mandatory. As long as the special GET/PUT buffer header for fonts is present, other groups can be used. EOU automatically loads 49 fonts. See /dd/sys/fontlist.txt for a list of the fonts, their descriptions and which size characters they are.

Parameters:

path	The route to the window in which you want to use an alternate font.
group	The group number of the buffer containing the font to use.
buffer	The number of the buffer containing the font to use.

Examples:

```
RUN GFX2("FONT",280,2)
```

PROPSW Proportional space switch

Syntax: RUN GFX2([path,]"PROPSW","switch")

Function: Enables or disables the automatic proportional spacing of characters on graphic screens.

NOTE: Proportional fonts will display notably slower than a standard 8x8 font.

Parameters:

path The route to the window in which you want to use proportional character spacing.

switch Either OFF to turn proportional spacing off, or ON to turn proportional spacing on. The default setting of the switch is OFF.

Examples:

```
RUN GFX2("PROPSW","ON")
```

Sample Procedure:
This procedure produces a demonstration of the BASIC09 proportional spacing function.

```
PROCEDURE proport
DIM LINE:STRING
DIM LETTER:STRING[1]
DIM T,J,K,FLAG:INTEGER
RUN GFX2("DWEND")
RUN GFX2("DWSET",7,0,0,80,25,3,2,2)
RUN GFX2("CLEAR")
FLAG=1
FOR T=1 TO 12
READ LINE
FOR J=1 TO LEN(LINE)
LETTER=MID$(LINE,J,1)
IF LETTER<>"!" AND LETTER<>"#" THEN
PRINT LETTER;
ENDIF
IF LETTER="!" THEN
FLAG=FLAG*-1
IF FLAG>0 THEN
RUN GFX2("PROPSW","OFF")
ELSE
RUN GFX2("PROPSW","ON")
ENDIF
ENDIF
IF LETTER="#" THEN
PRINT CHR$(34);
ENDIF
NEXT J
```

```
PRINT
NEXT T
PRINT \ PRINT
END
DATA "This is a demonstration of"
DATA "!Proportional! Spacing! using"
DATA "BASIC09's GFX2 module."
DATA ""
DATA "!The quick brown fox jumped...!"
DATA "The quick brown fox jumped..."
DATA ""
DATA "Use the command"
DATA "!RUN GFX2(#PROPSW#,#ON#)!"
DATA "to turn proportional spacing on."
DATA "Use !RUN GFX2(#PROPSW#,#OFF#)!"
DATA "to turn proportional spacing off"
```

REVOFF Reverse video off

Syntax: RUN GFX2([path,]"REVOFF")

Function: Disables reverse video characters. Once set, reverse video remains in effect until you execute the reverse video off function.

Parameters:

path The route to the window in which you want to display reverse characters.

Examples:

```
RUN GFX2("REVOFF")
```

REVON Reverse video on

Syntax: RUN GFX2([path,]"REVON")

Function: Enables reverse video characters. Once set, reverse video remains in effect until you execute the reverse video off function.

Parameters:

path The route to the window in which you want to display reverse characters.

Examples:

```
RUN GFX2("REVON")
```

TCHARSW Transparent characters switch

Syntax: RUN GFX2([path,],"TCHARSW","switch")

Function: Enables or disables transparent characters. If enabled, on graphics screens it means that anything underneath text you PRINT will remain on the screen. If disabled, anything underneath text you print will be cleared to the currently selected background color. On hardware text screens, if enabled, the foreground color, blink and underline attributes are changed to your current settings, but the background color will be left alone.

Parameters:

path	The route to the window in which you want to use transparent characters.
switch	Either OFF to turn transparent characters off, or ON to turn transparent characters on. The default setting of the switch is OFF.

Examples:

```
RUN GFX2("TCHARSW","ON")
RUN GFX2("TCHARSW","OFF")
```

UNDLNOFF Underline characters off

Syntax: RUN GFX2([path,]"UNDLNOFF")

Function: Disables character underline. After you execute UNDLNON, all characters displayed are underlined until you execute UNDLNOFF. The default is UNDLNOFF.

Parameters:

path The route to the window where you want to use underline characters.

Examples:

```
RUN GFX2("UNDLNOFF")
```

UNDLNON Underline characters on

Syntax: RUN GFX2([path,]"UNDLNON")

Function: Enables character underline. After you execute UNDLNON, all characters displayed are underlined until you execute UNDLNOFF. The default is UNDLNOFF.

Parameters:

path The route to the window where you want to use underline characters.

Examples:

```
RUN GFX2("UNDLNON")
```

Mouse Handling Functions

Function	Description
Mouse	Return mouse information
OnMouse	Set mouse clicked/key pressed signal
SetMouse	Set mouse parameters

MOUSE Return mouse information

Syntax: RUN GFX2("MOUSE",valid,fire,x,y,[area,sx,sy])

Function: Returns Mouse packet information.

Parameters:

valid 0=data not valid (not interactive window)
fire Mouse fire button status.
1. =no buttons down,
2. =Button A down,
3. =Button B down,
4. =Both buttons down

X Scaled X coordinate (window relative)
y Scaled Y coordinate (window relative)

The following 3 parameters are optional and used for menu framed windows. All 3 must be present for a menu framed window in order to use the Multi-Vue menuing system.

area Window area type if WNSET has been used to make a menu framed window (with or without scrollbars).
1. =Inside the working area of the window
2. =Outside the working area, but in the menu region. This can be the top menu bar, or scroll bars (if in a framed scrollbar window)
3. =Off window entirely (mouse on different device window)

sx Unscaled X coordinate (full screen coordinate)
sy Unscaled Y coordinate (full screen coordinate)

NOTES: One can use the *valid* parameter to see if their program is the current interactive window (has control of the keyboard, mouse and screen). If *valid* returns 0 (meaning it is not the current interactive window), the program can pause and/or put itself to sleep for the remainder of the current clock tick (or use **OnMouse**). It can then monitor when it becomes the interactive process again, and then unpause and resume running normally. This can be used as an auto-pause in games if the player switches off of the game window, for example.

Examples:

```
RUN GFX2("MOUSE",validflag,button,x,y)
RUN GFX2("MOUSE",validflag,button,x,y,
areatype,rawx,rawy)
```

Sample Procedure:

```
PROCEDURE Mouse
DIM validflag,button,x,y,oldx,oldy:INTEGER
DIM buttons(3):STRING[12]
buttons(1)="Button 1"
buttons(2)="Button 2"
buttons(3)="Buttons 1&2"
RUN GFX2("DWEND")
RUN GFX2("DWSET",8,0,0,40,25,0,2,2)
RUN GFX2("SETMOUSE",$FF,$FF,1)
RUN GFX2("GCSET",$CA,1)
RUN GFX2("FONT",$C8,2)
oldx=-1
oldy=-1
LOOP
RUN GFX2("MOUSE",validflag,button,x,y)
EXITIF validflag=0 THEN
PRINT "Mouse off window"
ENDEXIT
IF button>0 THEN
RUN GFX2("CURXY",0,23)
PRINT buttons(button);" clicked while mouse at coordinate
";x;",";y;
RUN GFX2("EREOLINE")
ELSE
IF oldx<>x OR oldy<>y THEN
oldx=x
oldy=y
RUN GFX2("CURXY",25,0)
RUN GFX2("EREOLINE")
PRINT x;",";y
ENDIF
ENDIF
EXITIF button=3 THEN
PRINT "Both buttons pressed, exiting program"
ENDEXIT
ENDLOOP
END
```

ONMOUSE Set up mouse button and key pressed signals, optionally sleep until a mouse click or key press

Syntax: RUN GFX2("ONMOUSE",signal)

Function: Sets up key and mouse button presses to send a signal to the calling program. There are two ways that this function can be set up, depending on the signal number requested to be sent:

Requesting a signals between 1 and 255 will let the program continue running normally until a key or mouse button is pressed; it will then send the signal code to the calling program. However, there is a caveat; If you are running BASIC09 itself it will cause the program break out as if the BREAK key was hit, and the program will not be able to trap it properly. For specific signal numbers to work, the program must be PACK'ed first, and run through the run time package RUNB. Used in this way, the signal coming in will trigger any ON ERROR GOTO the program has set up, and the signal number will become the error code (ERR). Because of this, it is recommended that one chooses a signal number that is *not* a regular error number (either for NitrOS9/OS-9 or BASIC09), so a signal number between 100-182 is recommended.

Requesting a signal of 0 is a special request to put the calling program to sleep until a key or mouse button is pressed; the program will then be woken up. This way the rest of the system gets to run full speed while waiting for the user to click something. The program will resume execution on the line following the ONMOUSE function. The program can then check for a mouse click (with MOUSE) or key press (INKEY or a SS.Ready GetStat system call, and act appropriately. This option is usually used when waiting for a user to make a menu selection for Multi-Vue type programs, but can be used for other things as well.

Parameters:

signal 0=put program to sleep until a mouse click or key press.
1-255 =Signal sent to the calling program when a mouse button is clicked or a key is pressed. This can only be trapped by ON ERROR GOTO, and only when a program is PACK'ed and running through RUNB (not BASIC09). This allows a program to continue running doing other things without constantly manually monitoring for a mouse click or key press.

Examples:

`RUN GFX2("ONMOUSE",0)` Put program to sleep until a key is pressed or mouse button is clicked.

`RUN GFX2("ONMOUSE",100)` Set up an ERROR 100 to trigger when a key is pressed or mouse button is clicked. (RUNB only)

Sample Procedure:

See the sample program for GETSEL.

SETMOUSE Set mouse parameters (see NOTE below)

Syntax: RUN GFX2(scanrate,timeout,autofollowflag)

Function: Can set the mouse scan rate & mouse timeout, as well as turning mouse cursor auto-follow on or off.

Parameters:

scanrate	How many clock ticks (1/60th of a second) between physical mouse reads. (3 is a nice balance of smoothness and friendly multi-tasking, 1/20th of a second). A value of 255 indicates to leave the current setting alone.
timeout	How many clock ticks (1/60th of a second) after a button press before the mouse goes into "quiet" mode. Pressing a mouse button starts a number of timers for both mouse buttons (time in current state, time in last state, to keep trick of multiple clicks). This timeout setting is how long before the system resets the counters while waiting for another mouse click. Once it hits this value all timers are cleared and shut off (taking less CPU overhead) until the next mouse click. A value of 255 indicates to leave the current setting alone.
autofollowflag	0=Turn auto-follow mouse off, 1=turn auto-follow mouse on. On means the operating system will handle all mouse position and drawing on the screen automatically.

Examples:

```
RUN GFX2("SETMOUSE",3,120,0)
RUN GFX2("SETMOUSE",255,255,1)
```

Sample Procedure:
See the sample program for GETSEL.

Music/Miscellaneous Functions

Function	Description
ID	Return process ID
Tone	Play a tone

ID — Return process ID

Syntax: RUN GFX2("ID",idnum)

Function: Returns the process ID # of the current process. Useful for reserving GET/PUT buffers unique to the caller's process, so as to not interfere with other processes currently running.

Parameters:

idnum The process ID #

Examples:

```
RUN GFX2("ID",idnum)
```

TONE — Play a tone through the speaker

Syntax: RUN GFX2("TONE",frequency,duration,volume)

Function: Plays a single tone through the speaker. NOTE: Unlike almost all other GFX2 functions, tones will play no matter which window is the active one.

Parameters:

frequency Tone frequency (1-4095)
duration Duration (1-255) (measured in 1/60th of a second increments).
volume Volume (1-63)

Examples:

```
RUN GFX2("TONE",500,60,63)
RUN GFX2("TONE",2000,1,31)
```

Chapter 10

BASIC09 Quick Reference

This chapter contains a quick reference of all BASIC09 commands, statements, and functions. It includes commands for programming, editing, and debugging, as well as the Command mode commands.

The following chart lists all BASIC09 keywords that you can use in a procedure.

Statements and Functions

Command	Description
ABS	Returns the absolute value of a number.
ACS	Calculates the arccosine of a number.
ADDR	Returns an integer value which is the absolute memory address of a variable, array, or structure in a process's address space.
AND	Generates the logical AND of two boolean values.
ASC	Returns the ASCII code of the first character in a string.
ASN	Calculates the arcsine of a number.
ATN	Calculates the arctangent of a number.
BASE	Sets the lowest array or data structure subscript in a procedure to either 0 or 1.
BYE	Ends execution of a procedure and terminates BASIC09.
CHAIN	Executes a module, passing arguments if appropriate.
CHD	Changes the current data directory.
CHR$	Returns the ASCII character represented by a specified integer.
CHX	Changes the current execution directory.
CLOSE	Deallocates the specified path to a file or device.
COS	Calculates the cosine of a number.
CREATE	Opens a path and establishes a new file on disk.
DATE$	Returns the computer's current date and time.
DEG	Causes BASIC09 to calculate angles in degrees.

Command	Description

DATA	Stores data in a procedure to be accessed by the READ statement.
DELETE	Deletes a file from disk.
DIM	Declares simple variables, arrays or complex data structure for size and type.
DO	See WHILE/DO/ENDWHILE.
ELSE	See IF/THEN/ELSE/ENDIF.
END	Terminates execution of a procedure. Returns to the calling procedure or to BASIC09's command mode. Displays the specified text.
ENDEXIT	See EXITIF/ENDEXIT.
ENDIF	See IF/THEN/ELSE/ENDIF.
ENDLOOP	See LOOP/ENDLOOP.
ENDWHILE	See WHILE/DO/ENDWHILE.
EOF	Tests for the end of a disk file.
ERR	Returns the error code of the most recent error.
ERROR	Generates the specified error.
EXITIF/ ENDEXIT	Tests conditions in a loop. The procedure exits the loop if the condition is true.
EXP	Calculates e (2.71828183) raised to the specified value.
FALSE	A boolean function that always returns FALSE.
FIX	Rounds a real number and converts it to an integer.
FLOAT	Converts a byte or integer value to a real number.
FOR/NEXT	Creates a program loop of a specified number of repetitions.
GET	Reads an element or a data structure from a binary file or a device.
GOSUB/ RETURN	Transfers program control to a specified subroutine. RETURN sends execution back to the calling routine.
IF/THEN/ELSE/ ENDIF	Evaluates an expression and performs an operation if the conditions are met. Including ELSE causes an alternate operation if the conditions are false.
INKEY	Stores the character of a keypress in a string variable.

Command	Description
INPUT	Causes a procedure to accept input from the keyboard or other specified device.
INT	Returns the largest whole number less than or equal to the specified value.
KILL	*Unlinks* a procedure. (Removes it from BASIC09's directory.)
LAND	Performs a bit-by-bit logical AND on two-byte, or integer, values.
LEFT$	Returns the specified number of characters, from the leftmost portion of a string.
LEN	Returns the length of the specified string.
LET	Assigns a value to a variable.
LNOT	Performs a bit-by-bit logical NOT function on two-byte, or integer, values.
LOG	Calculates the natural logarithm.
LOG10	Calculates a base 10 logarithm.
LOOP/ ENDLOOP	Establishes a loop. Use EXITIF and ENDEXIT to test the loop and exit when a specified condition is true.
LOR	Performs a bit-by-bit logical OR on two-byte, or integer, values.
LXOR	Performs a bit-by-bit logical EXCLUSIVE OR on two-byte, or integer, values.
MID$	Returns the specified number of characters, beginning at the specified position in a string.
MOD	Returns the modulus (remainder) of a division operation.
NEXT	See FOR/NEXT.
NOT	Returns the logical complement of a boolean value.
ON ERROR/ GOTO	Traps errors and transfers control to the specified line number. ON ERROR without the GOTO turns off error trapping. PACKed procedures can only trap signals with ON ERROR GOTO.
ON/GOSUB	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execution

transfers to the selected line.

Command	Description
ON/GOTO	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execute jumps to the selected line.
OPEN	Opens an IO path to an existing file or device.
OR	Performs a logical OR on two boolean values.
PARAM	Describes the parameters a called procedure expects from a calling procedure.
PAUSE	Suspends execution of a procedure, and enters the Debug mode.
PEEK	Returns the byte value of a memory address.
PI	Represents the constant 3.14159265.
POKE	Stores a byte value at a specified memory address.
POS	Returns the current character position of the print buffer.
PRINT or **?**	Sends the specified characters or values to the display. The ? shorthand can be used with any of the following PRINT keywords as well.
PRINT USING	Sends characters or values to the display, using the specified format.
PRINT #	Sends the specified characters or values to the specified path.
PRINT # USING	Sends characters or values to the specified path using the specified format.
PUT	Writes data to a random access file.
RAD	Causes BASIC09 to calculate angles in radians.
READ	Accesses data from procedure DATA lines or from files or devices.
REM or **!**	Indicates that the following characters in a procedure line are comments and are not to be executed. Also use (* *), or (*.
REPEAT/UNTIL	Establishes a loop that executes until the specified condition is met.
RESTORE	Restores the DATA pointer to the first data item or to a specified line.
RETURN	See GOSUB/RETURN.
RIGHT$	Returns the number of characters specified, from the rightmost portion of a string.
RND	Returns a random number from a specified range.

Command	Description
RUN	Calls another procedure for execution.
SEEK	Changes the file pointer address.
SGN	Determines the sign of a number.
SHELL	Calls an NitrOS-9 command or program for execution.
SIN	Calculates the sine of a specified value.
SIZE	Returns the number of bytes assigned to a variable, array, or complex data structure.
SQ	Calculates a value raised to the power of two.
SQR/SQRT	Calculates the square root of a positive number.
STEP	Sets the size of increment in a FOR/NEXT loop.
STOP	Terminates the execution of all procedures and returns to the BASIC09 Command mode.
STR$	Converts numeric data to string data.
SUBSTRING	Returns the starting position of a sequence of characters in a string.
SYSCALL	Executes an NitrOS-9 System Call.
TAB	Begins a print operation at the specified column.
TAN	Calculates the tangent of a value.
TRIM$	Strips trailing spaces from the specified string.
TRON/TROFF	Turn the trace mode on and off.
TRUE	Returns the boolean value of TRUE.
TYPE	Defines a new data type.
UNTIL	See REPEAT/UNTIL.
USING	See PRINT USING.
VAL	Converts a string to an integer.
WHILE/DO/ ENDWHILE	Executes a loop as long as a specified condition is true.
WRITE	Writes data in ASCII format to a file or device.
XOR	Performs a logical EXCLUSIVE OR on two boolean values.

Commands by Type

Statements

BASE 0	DIM	GOSUB	OPEN	RETURN
BASE 1	ELSE	GOTO	PARAM	RUN
BYE	END	IF/THEN	PAUSE	SEEK
CHAIN	ENDEXIT	INPUT	POKE	SHELL
CHD	ENDIF	KILL	PRINT	STOP
CHX	ENDLOOP	LET	PUT	TROFF
CLOSE	ENDWHILE	LOOP	RAD	TRON
CREATE	ERROR	NEXT	READ	TYPE
DATA	EXITIF/THEN	ON ERROR/GOTO	REM	UNTIL
DEG	FOR/TO/STEP	ON/GOSUB	REPEAT	WHILE/DO
DELETE	GET	ON/GOTO	RESTORE	WRITE

Transcendental Functions

ACS	COS	LOG10	SIN
ASN	EXP	PI	TAN
ATN	LOG		

Numeric Functions

ABS	LAND	MOD	SQ
FIX	LNOT	RND	SQR
FLOAT	LOR	SGN	SQRT
INT	LXOR		

String Functions

ASC	LEFT$	RIGHT$	TRIM$
CHR$	LEN	STR$	VAL
DATE$	MID$	SUBSTR	STR
INKEY			

Miscellaneous Functions

ADDR	FALSE	SIZE	SYSCALL
EOF	PEEK	TAB	
ERR	POS	TRUE	

Data Types

The following list shows the BASIC0S data type you can specify when defining a variable.

Type	Function
BOOLEAN	Returns TRUE or FALSE.
BYTE	Specifies that a numeric variable is to store single-byte values.
INTEGER	Specifies that a numeric variable is to store integer (two-byte) values.
REAL	Specifies that a numeric variable is to store real (five-byte) values.
STRING	Specifies that a variable is to store ASCIT characters.

Types of Access for Files

You can use the following parameters with the CREATE and OPEN commands. Check the individual commands for information on which parameter to use with which command.

Parameter	Function
DIR or **[ENTER]**	Lets BASIC09 access a directory-type file for reading. Do not use with UPDATE or WRITE.
EXEC	Lets BASICO9 access the current execution directory rather than the current data directory.
READ	Sets the file access mode for reading.
WRITE	Sets the file access mode for writing.
UPDATE	Sets the file access mode for both reading and writing.

Command Mode

The following chart lists the commands available from the BASIC09 Commands mode:

Command	Function
$	Calls the shell command interpreter to execute an NitrOS-9 command. Type **EX** to return to BASIC09.
$*command*	Tells BASIC09 to execute the specified NitrOS-9 command(s) or program(s). Upon completion it immediately returns to BASIC09. To run more than one command, separate each command with a semi-colon (';').
BYE or **[CTRL][BREAK]**	Returns you to the NitrOS-9 system or to the program that called BASIC09.
CHD	Changes the current data directory.
CHX	Changes the current execution directory.
DIR	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or **E**	Enters the procedure editor/compiler mode.
KILL	Removes one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a file into the workspace.
MEM	Displays current workspace size or reserves a specified amount of memory for the workspace.
PACK	Performs a second compilation and stores the resulting file in the execution directory. **NOTE:** Source code must be saved before using PACK or it will be lost!
RENAME	Changes a procedure's name.
RUN	Causes a procedure to execute.
SAVE	Writes one or more procedures to disk.

Edit Commands

The following chart lists the commands available from the Edit mode:

Command	Function
[ENTER]	Moves the edit pointer to the next line.
+*num*	Moves the edit pointer forward a specified number of lines.
+*	Moves the edit pointer past the last line.
-*num*	Moves the edit pointer back a specified number of lines.
-*	Moves the edit pointer to the first line.
text	A space followed by text inserts an unnumbered line before the current line.
line	Typing a line number with or without text following it inserts the line into the procedure.
line **[ENTER]**	Moves the edit pointer to the line *line*.
c/*str1*/*str2*/	Changes the text *str1* to the text *str2*.
c*/*str1*/*str2*	Changes all occurrences of *str1* to *str2*.
d	Deletes the current line.
d*	Deletes all the lines in the current procedure.
d*num*	Deletes *num* lines including the current line in the current procedure.
l	Lists the current line.
l*	Lists all the lines in the current procedure.
l*num*	Lists *num* lines including the current line in the current procedure.
q	Terminates the edit session.
r	Renumbers lines from the first line number, in increments of 10.
r*	Renumbers all numbered lines in increments of 10. The first line number is 100.
r *line*	Renumbers lines from *line* in increments. of 10.
r *line num*	Renumbers lines from *line*, in increments of *num*.
s/*str*	Searches for the first occurrences of *str*.
s*/*str*	Searches for all occurrences of *str*.

Debug Commands

The following table lists all the Debug commands and what they accomplish:

Command	Function
$	Calls the shell command interpreter to execute an NitrOS-9 command. Type **EX** to return to BASIC09.
$*command*	Tells BASIC09 to execute the specified NitrOS-9 command(s) or program(s). Upon completion it immediately returns to BASIC09. To run more than one command, separate each command with a semi-colon (';').
BREAK	Sets a breakpoint at the specified procedure.
CONT	Causes procedure execution to continue.
DEG/RAD	Selects either degrees or radians as the unit of angle measurement for trigonometric functions.
DIR	Displays the procedures in the workspace.
Q	Leaves the Debug mode for the System mode.
LET	Assigns a new value to a variable.
LIST	Displays a source listing of the suspended procedure.
PRINT *var*	Displays the value of the specified variable.
STATE	Lists the *nesting* order of all active procedures.
STEP *num*	Causes execution of the suspended procedure in specified increments.
TRON/TROFF	Turns the trace function on and off.