Learning to Program with the Cybiko Handheld Computer Using B2C BASIC

Learning to Program with the Cybiko Handheld Computer Using B2C BASIC

Gregory Smith
Thinkable.Us

“Say yes when you can, and no when you must”
- Unknown

Dedicated to my children, who inspire me daily:
Amber & Heather

Contents

Preface

Teen Computer

Donald Wisniewski and David Yang created the Cybiko as a portable computer for tweens (ages 10-12) and teens (ages 13-17). It was designed to fill a niche left between the Palm Pilot and Nintendo Game boy.

Look & Feel

The original Cybiko looks like a walkie-talkie encased in a transparent shell from one of five colors (black, clear, yellow, blue, and purple – and a promotional red). It has an array of white buttons comprising a QWERTY-style keyboard, a joystick pad, and some function keys. The original, older model (released in April 2000) had a power slide switch on the side, which was later replaced by an "Esc" key.

The new Cybiko Xtreme (CyX2) is very different from the original Cybiko. Released in September of 2001, the CyX2 body has a trim feel to it compared to the Cybiko. It is roughly the same size as the original Cybiko. It's molded as all one unit and shaped like a figure 8 (curved edges, wide at the top, narrow waist, and wide near the bottom). In this way it resembles the m100 model of the Palm Pilot. And like the m100 series, there are skins that cover the silverized CyX2 body. There are also separate panels for the areas around the keyboard, arrow keys, and the Enter pad. Red, green, blue, and zebra colors are available. You can even mix and match colors for a unique color scheme.

The CyX2 keyboard has larger keys and they are of a solid black rubberized material. The arrow keys resemble the Game Boy keys and are separate from the ESC key. In a brilliant and very welcome move, there is an ON/OFF key situated in the center of the keyboard. One drawback to the new, larger keyboard is that Cybiko removed the top row of number keys (1-9). To type numbers into the unit you must use the FN key to 'shift'. The F1-F7 keys still exist at the top of the unit.

The CyX2 antenna is no longer the 6" long rod rising from the back of the unit. Instead, a slim side-antenna (again resembling the Palm Pilot VII antenna). CyX2 sports a mini USB port at the bottom of the case. This USB port will be used to download games. It also doubles as the battery charger connection. A supplied cable connects the USB port to a power adapter. The display is the same as on the Cybiko: 160x100 pixels, 4-color grayscale. A slot for additional devices is now at the top of the unit.

Donald Wisniewski & Cybiko Inc.

The creators of the Cybiko – Donald Wisniewski (President of Cybiko, Inc.) and David Yang started the company in early 1999. They recognized a need in the teen marketplace for a handheld wireless entertainment device that teens could call their own.

Wisniewski (born in 1960) is a former executive of V-Tech (another high-tech firm in the business of creating devices for young people). He holds a BS in mechanical engineering from Purdue University. "We want to create a social environment where teens can chat and interact," he said in a recent interview, "We see a huge opportunity with 12- to 16-year olds."

The Cybiko was released in April of 2000 with a price tag of $170. At that time, New York-based Cybiko had 85 engineers who had created the device. As of January 2001 Cybiko had a staff of around 170 Russian-based software developers and retailed for about $99. They had successfully delivered a new application daily from about January 2000 to July 2001. The product was first introduced at a number of toy and electronics retailers like FAO Schwarz, Virgin Records, CompUSA, Babbage's, and Software Etc. Now, it is available at over 9000 locations across the USA and online at www.cybiko.com. Cybiko is privately held and owns all of the intellectual property.

The CyX2 was released in September 2001. The game-a-day policy has changed to a trickle of just one game per week.

Teens and Cybiko

There are 30 million teens in the United States, 65 percent have a computer in the home. 45 percent have Internet access and none of them had a mobile wireless product. That is a market of 13.5 million people. The Cybiko is aimed at Nine to 20-year-olds. They felt interconnectivity and communication was a key feature left out of other devices for teens. "We wanted to create a social environment where teens can chat and interact," said Wisniewski.

And they wanted to create a device that was more than a game machine – so they included single and multiplayer wireless games, support applications like the planner, email, Friend Finder, and a way to chat with anyone in the network. "I've seen users as young as five and old as 55," Wisniewski said in a recent interview.

Technical Details

*(Note: See the glossary for a description of new terms)

The operating system for the Cybiko is called CyOS. It is now (Sept/2001) up to version 1.3.57. The original Cybiko (v1.2) advertised a mere 256KB of RAM. Currently the memory is divided into 2 segments : 512KB of RAM and 512KB or Flash RAM – for a grand total of 1MB. Most of the memory is used up by the Cybiko Operating System (CyOS). Only about 300KB is available for the user. The memory is supposed to be upgradeable to 16MB, but this number is different depending on the published source. (There is a 1MB expansion module but the Cybiko can take advantage of the MP3's memory which can be as large as 64MB).

The CyX2 boasts 1MB of RAM and 1MB or Flash RAM. With an additional 500KB of ROM, the CyX2 weighs in at 2.5MB of memory.

The Cybiko's display is 160 x 100 pixels of monochrome LCD (actually, its 4-color gray scale). Its Microprocessor is a 32-bit 11-MHz processor manufactured by Hitachi (this is roughly equivalent in capability to an Intel 386 chip – the heart of the original IBM PC) The CyX2 has a 22-MHz processor.

The Cybiko broadcasts on 30 digital channels over frequencies between 902 and 928 MHz with a range of up to 300 feet. There has been a lot of talk on the Internet about improving the range with home-brew antennae and so on, but for all practical purposes 300 feet is as far as the Cybiko can broadcast. Wireless email is possible with the CyWIG software on the PC. Users can send files (including music, games and applications) between two Cybikos. While Cybiko users can chat and game with other Cybikos within a range of 300 ft. They can also transmit through each other to extend the range, similar to a re-transmitter.

The original Cybiko has a small keyboard allows for data entry. The keys are arranged in the traditional QWERTY layout, but are smaller than those used on the popular RIM pager devices. A stylus is provided to allow easier typing. There is a small joystick pad with up, down, left, and right keys. Seven pre-programmed function keys line the top of the unit. And the help, enter, select, tab, and del keys are along the right side. In fact, you will mostly use the arrow keys and the enter key.

The CyX2 keyboard has black letters that are larger than its predecessor.

The expansion slot is located at the base of the Cybiko. It is able to take an optional MP3 player. Also, a memory card with 1MB of memory is available. The CyX2 expansion slot is at the top of the unit.

Two rechargeable nickel-metal hydride batteries that last for 8-10 hours of regular use supply power. The unit can vibrate whenever a similar machine is in range.

An RS232 cable connects it to a PC, which allows users to download new games and applications from the Internet. It is also possible to write programs that access the RS232 port.

The CyX2 uses a USB port to communicate with the outside world.

Chapter 1 : Introducing BASIC & B2C

Welcome to programming in B2C. B2C stands for Basic-to-C compiler. With B2C you can create very simple to very complex programs (including video games) with a minimum of effort. In the following pages you will find a step-by-step introduction to programming in B2C. To gain the maximum benefit from this tutorial you are encouraged to do each step one by one in order. When you are done, you will know enough to write any program you can imagine – and we know you have a wild imagination!

BASIC is an acronym. It stands for Beginner's All-purpose Symbolic Instruction Code. It was invented in the 1960's as a simplified programming language for people of all types. The original language had a line number for each line of code. This helped to order the statements and make it easier for the newcomer to change something on a particular line. The purpose of the BASIC language is to make programming code more like English so that it can be easily learned by the non-computer professional.

B2C is based on the CyBasic2 language by Cybiko, Inc. Unlike the 'C' programming language, CyBasic2 is interpreted. Languages like 'C' are compiled, meaning the instructions are transformed into machine code. CyBasic2 is interpreted at run time. It is never converted into machine code. So, it is slower than 'C'. B2C is a compiler – it transforms B2C code into C that is then compiled into a fast-running application.

CyBasic2 was a fairly limited language. It had the ability to get input from the user, process that input, and print the processed results to the screen. The If command allowed CyBasic2 to handle conditionals and the For command allowed it to handle looping. There are only a handful of graphic functions including Point (for setting a pixel) and Line (for drawing a line).

B2C on the other hand has commands to work with bitmaps directly. Anything you can do in C you can also do in B2C, only with B2C, it's easier. You can create real video games with B2C, something CyBasic2 was never designed to do.

B2C also embeds passwords into the resultant program. The program will run for 60 seconds before the password screen pops up. The password can be downloaded by running the included KeyGen.app program.

Follow this link for more information on the origins of BASIC:
http://www.digitalcentury.com/encyclo/update/BASIC.html

Chapter 2 : Downloading B2C

B2C is available as a download at www.DevCybiko.com/basiccompiler.html. It is currently in version 3 (B2Cv5.zip). When you download it, you must un-zip the file using a Zip tool (like PKZip or ZipMagic). Be sure to place B2Cv5 in a directory that is easy to use (like C:\B2Cv5, for example).

The next step is to download and install the Cybiko SDK from www.cybiko.com/devsarea (or www.devcybiko.com/basiccompiler.html). You will need Version 2.0 or better of the Standard or Professional SDK. (The Professional SDK costs $49.95. It is not necessary to purchase the professional version, the Standard works just fine).

DO THIS
If you want to cut to the chase and just rush through the tutorial, look for this box. Do the things in this box to learn how to program in B2C faster. If you didn't understand what happened in this box, then re-read the section above it.

Chapter 3 : Editing Your First Program

Introduction

The process for any B2C program is simple: Edit, Compile, Download, Run, and Repeat. In this chapter we'll look at the first part: Editing.

There are many good editors for use on the PC. Every programmer has a favorite editor. Here, we will discuss the Notepad editor for Windows, and the Edit program for MS-DOS. You will also be introduced to the MS-DOS Command Prompt, since you will spend much of your time there.

Notepad

The Notepad editor is a bare-essentials editor, which is precisely what we need for editing B2C programs. You can run Notepad by selecting the "Start" menu and pulling-right on "Accessories". Pull-down to the "Notepad" icon. Here is a summary of the Notepad menu options:

Menu Option	Keyboard Equivalent	Comments
File->New	Alt-F-N	Create a new, empty, text file
File->Open	Alt-F-O	Open an existing file
File->Save	Alt-F-S	Save the current file
File->Save As	Alt-F-A	Save the current file with another name
File->Page Setup	Alt-F-T	Print out setup
File->Print	Alt-F-P	Print the file
File->Exit	Alt-F-X	Exit Notepad
Edit->Undo	Ctrl-Z	Undo edits
Edit->Cut	Ctrl-X	Cut current selection
Edit->Copy	Ctrl-C	Copy current selection
Edit->Paste	Ctrl-V	Paste buffer
Edit->Select All	Alt-E-S	Select entire file
Edit->Time/Date	F5	Insert Current Time/Date into file
Edit->Word Wrap	Alt-E-W	Wrap words at the end of the line (or let lines go offscreen)
Edit->Set Font	Alt-E-F	Set the font (default : courier)
Search->Find	Alt-S-F	Find text
Search->Find Next	F3	Find text again
Search->Replace	Alt-S-R	replace text (only on Windows NT)
Help->Help Topics	Alt-H-H	Help information on Notepad
Help->About	Alt-H-A	Copyright information

Run Notepad and type the following…

```
print "hello world"
dim b
input b
```

Now exit the Notepad selecting the File->Exit menu and answer "Yes" when it asks you if you want to save the file. Use the filename "hello.b2c".

MS-DOS and Edit

The MS-DOS Edit command is a more powerful alternative to the Notepad editor. It offers a more professional set of features than Notepad. As a programmer, these features give you an easier path from your ideas to your code. The faster you can type your ideas into the editor, the more ideas you can code. So what features do we need in an editor? Here is a list of the bare essentials:

- open file

- save file
- enter and edit text
- cut, copy and paste of text
- tab or indent
- search
- search & replace

You may desire other features, but I find these to be the minimum. Notepad (for Windows 9x) does not have a search & replace and line numbering.

MSDOS as Your Working Environment

The MS-DOS (Microsoft Disk Operating System) Prompt is still delivered with all MS Windows operating systems. This is a viable working environment for our needs.

DO THIS
To run the MS DOS Command Prompt, select the Start menu and Programs. Pull right and find MS DOS Prompt (or Command Prompt) in the menu.

You may want to modify some of the properties of the MS-DOS window. For example, the default setting is for a 24-line display. I like more lines of text when using the text editor. To change the properties, click the left mouse button over the MS-DOS logo in the upper-left corner of the window - this will display a menu. Select "Properties" at the bottom of the menu.

Here are the settings I like to use for MS-DOS:

Tab Selection	Setting	Default	Recommended
Program	-keep all defaults-	-keep all defaults-	-keep all defaults-
Font	Bitmap Only	Both Font Types	Bitmap Only
	Font Size	Auto	8x12
Memory	-keep all defaults-	-all set to Auto-	-keep all defaults-
Screen	Usage	Window	Window (not Full Screen)
	Initial Size	Default	50 lines
	Window	Display Toolbar (checked)	Do Not Display Toolbar
		Restore Settings (checked)	Restore Settings (checked)
	Performance	Fast ROM Emulation (checked)	Fast ROM Emulation (checked)
		Dynamic Memory Allocation (checked)	Dynamic Memory Allocation (checked)
Misc	-keep all defaults-	-keep all defaults-	-keep all defaults-

MS-DOS Commands

There are relatively few MS-DOS commands that you will need to know in this tutorial. Fortunately, the majority of tasks can be handled in Windows. Nevertheless it is useful to know how to do certain, fundamental operations. The following table summarizes them:

Command	Syntax	Remarks
Del	del filename.ext del *.ext	deletes one or more files (*=wildcard characters)
Copy	copy file1.ext file2.ext	copies file1 to file2
Cd	cd dirname cd ..	change directory from one place to another change directory to parent directory
batch file	filename.bat	a list of MS-DOS commands in a file with the extension .bat. This will be executed when you type in the name of the file (like make.bat in Step1.app)
Dir	dir dir *.ext	- list all files in current directory - list only files with certain extension

Introducing MS-DOS Edit

DO THIS
In the MS-DOS Prompt, type `cd C:\…\B2Cv5  (where … is the path to B2Cv5)`

DO THIS

Next type "edit" and when the blue screen pops up, type in the following…
 	
`print "hello world"`
`dim b`
`input b`
 	
Now exit the editor by typing the ALT-F-X commands (or choosing File->Exit with the mouse) and answer "Y" when it asks you if you want to save the file. Save the filename as "hello.b2c"

MS-DOS Edit Commands

MS-DOS Edit is a very straightforward editor with a few hidden options. It is "mouse-enabled," meaning that if you click on the menu bar, the expected Windows-like thing will happen. Clicking on some text in the Editor window will move the cursor to that position. You can drag the mouse over text and it will select the text.

If you would rather, you can use the keyboard for menu operations by holding down the ALT key (at the bottom of the keyboard, next to the space bar). When you hold down the ALT key, the menu "lights up" and you may press the highlighted character to drop-down the selected menu. For example, selecting ALT-F drops-down the File menu. Also, holding down the SHIFT key and moving the arrow keys will select text for cut and paste operations.

In dialog boxes, the TAB key usually will move you from field to field. The arrow keys will move you up, down, left, and right. If you select text in the Editor window and hit the TAB key, the selected region will be indented one tab stop (usually 8 characters). Holding down the SHIFT key and hitting TAB will "outdent" one tab stop.

MS-DOS Edit Menu commands

Menu Option	Keyboard Equivalent	Comments
File->New	Alt-F-N	Create a new, empty, text file

File->Open	Alt-F-O	Open a file that has already been created
File->Save	Alt-F-S	Save the currently displayed file
File->Save As	Alt-F-A	Save the currently displayed file with another name
File->Close	Alt-F-C	Close the current file and its window
File->Print	Alt-F-P	Print the currently displayed file
File->Exit	Alt-F-X	Exit the Editor
Edit->Cut	Ctrl-X	Delete the currently selected text and copy it to the buffer
Edit->Copy	Ctrl-C	Copy the currently selected text to the buffer
Edit->Paste	Ctrl-V	Insert the buffer into the currently selected file
Edit->Clear	Del	Delete the currently selected text (but don't copy it to the buffer)
Search->Find	Alt-S-F	Open the Find Dialog Box and search for the first occurrence
Search->Repeat Last Find	F3	Repeat the last find command
Search->Replace	Alt-S-R	Open the Search/Replace Dialog box
View->Split Window	Ctrl-F6	Split the current window in two horizontally
View->Size Window	Ctrl-F8	Begin resizing the split windows. Use the up and down arrow keys
View->Close Window	Ctrl-F4	Close the currently selected split window, restore to a single window pane
View->n	Alt-n	where n=1-9 - make the selected file the currently selected file
Options->Settings	Alt-O-S	Display the Settings dialog box
Options->Colors	Alt-O-C	Display the Colors dialog box allowing you to customize the colors to your preferences
Help->Commands	Alt-H-C	Display a listing of keyboard commands
Help->About	Alt-H-A	Display copyright information

The File->Open and File->Save As menu options will display a dialog box with ...

- Filename input field - can hold wildcard (*) patterns or filenames
- Current Working Directory
- Filename listbox - showing all files matching the pattern in the Filename input field
- Directories listbox - showing the parent dir (..), any directories, and other devices ([-A-], [-C-], [-D-], etc...)
- Open Read Only Checkbox - select this if you don't want to modify the file when you open it (either click on it with the mouse, or tab over to it and hit the space bar)
- Open Binary Checkbox - select this if you want to open files with other than ASCII data (like .app files)
- Line Width - for binary files only, the number of characters per line to display

The File-Print menu option will give you the option of printing selected text or the entire file.

The Search->Find menu option will display a dialog box with ...

- Find What - the text string you want to find
- Match whole word only checkbox - use this to find the search string as a word (surrounded by white space - space characters, tabs, newlines)
- Match case - use this to force upper and lower case letters to match

The Search->Replace menu option will display a dialog box with ...

- Find What - the text string you want to replace
- Replace With - the text string you want to replace the search string with
- Match whole word only checkbox - use this to find the search string as a word
- Match case - use this to force upper and lower case letters to match

- Replace Button - replace a single occurrence of the search string
- Replace All - replace all occurrences in the currently selected file

The View->Split option splits the current window into 2 panes. You are free to open a second, different file into the second window. You may resize the panes by selecting View->Size Window and moving the "center bar" up and down with the arrow keys. Or, you can use the mouse pointer to grab the "center bar" and drag it to the size you like. Selecting View->Close Window will return the Editor to single-pane viewing.

The Options->Settings will display a dialog box with ...

- Tab Stop (defaults to 8, but I recommend 4) - the number of spaces to indent when TAB is entered.
- Colors - allows you to change many color parameters in the Editor. I recommend keeping the defaults. If you get lost, you can always come here and click on "Defaults".

The Help->Commands menu option displays a dialog box with a listing of many "hidden" keyboard commands. I won't repeat their descriptions here. Most of them are obvious (like Home to return to the beginning of a line). But there are a few surprises (like CTRL-Y to delete a line). Use the Page-Up and Page-Down keys to scroll through the list.

Chapter 4 : Compiling Your First Program

B2C programs require a set of supporting files to create an application. Fortunately for you the B2Cbuild process handles this for you. You may never have to look at these files.

DO THIS
At the MS DOS command prompt type "build hello.b2c". If any errors occur you will see the word "ERROR", but if all goes well you should see a line which says something like "Done 7 files." In the event of an error, examine the error line. It will show the source file followed by a line number in parenthesis, followed by the error. For Example: hello.b2c(2) – Parse error Indicates an error in line two of hello.b2c. A parse error is some sort of spelling error.

Chapter 5 : Downloading and Running Your Application

If you have installed CyberLoad on your PC, then you can download your application easily.

First, verify that your Cybiko is connected to the computer in the usual way.

DO THIS
Just double-click on the file "hello.app" in the Windows Explorer (file browser). The file will be automatically downloaded to your Cybiko

Now that you have downloaded your application to the Cybiko, find it and run it. It will have the "B2C" logo for an Icon, and may have "Tardis Adventure" as the text (unless you changed it in chapter 5). Run it. It will say "hello world" and wait for you to click on the Enter key.

Chapter 6 : Programming Basics : Input/Process/Output

Input, Processing, Output

All computer programs have three elements – Input, Processing, and Output. Input is information (or data) fed into the computer. Input can be in the form of a textual string of characters from the keyboard, a mouse movement, or a data file. Output is information (or data) sent out of the computer. Output can be in the form of text displayed on the computer screen, mouse movement, graphics, or writing to a data file. Finally, Processing is any action performed on the input data to generate the output data.

Input

B2C has two commands to get data from the user and into the Cybiko computer. The major one is called 'Input' and the other one is called 'key' (we'll discuss key later). Input takes data from the keypad and inserts it into a variable (we'll discuss variables more in the next chapter). The Input statement looks like this:

```
input a
```

The 'a' in the input statement is the name of a variable, which receives the data.

In B2C the input statement accepts a prompt string. This string is displayed on the screen before the user enters their value:

```
dim name[32]
input "Enter your name", name
```

Output

The command for output to the Cybiko screen is 'Print'. The Print statement takes one or more variables and character strings (separated by commas) and displays them on the screen. In its default mode, the screen holds 7 lines of about 23 characters each (The FONT command, as we'll later see, can change this).

```
print "The value of a is", a
```

The stuff between the quotes (") is called a "Literal String". This is a string of letters, digits, and other characters that you want displayed on the screen. Here, the variable 'a' will be displayed. It is important to realize that 'a' is not displayed, but rather, it's value.

Also note that typing this statement into B2C alone will result in an error. You will need to Dimension (Dim) any variables before using them. Dim is covered in the next chapter.

Processing

Nearly everything else in the B2C set of commands can be considered processing. The simplest sort of processing is assignment. The assignment operation is the equal sign '='. You can assign a value to a variable like this:

```
a = 1
```

We're giving the variable 'a' a value of one. Consider the next examples…

```
a=1
b=a+1
print b
```

Here, 'b' is given the value 'a+1'. In this case the variable 'a' is replaced with the *value* of 'a' (which is one). Hence, b=1+1 or b=2. So, the value of 'b' is two.

Comments

All good computer languages have a way to document the *code* (programmer lingo for the statements comprising a program) in the program itself. Commenting a program is good practice in the event that you want to share your program with someone else. Commenting is also good as a way of reminding yourself what you intended when you wrote the code to begin with. It is good practice to have one line of commentary for each line of code – on average. In B2C, comments are identified by a single-quote mark (also called the 'apostrophe' mark).

```
' demo program for chapter 7
dim a
dim b
a=1   'set a to one
b=a+1   'set b to one greater than a
Print b    'show the user the value of b
```

As we'll see in the upcoming example program, it is also useful to name your variables in a self-documenting way. If you are summing 4 grades, name the variables grade1, grade2, grade3, grade4, sum (the sum of the grades) and avg (the average of the grades). While variable names like 'a' and 'b' are short and easy to type, they are also cryptic and hard to remember.

Upper and Lower Case

B2C does not care about upper or lower case. A varaible name written in uppercase (A) in one place can be referred to later in lowercase (a). The names of statements in B2C are also not case sensitive. So Print and PRINT and print are all the same function.

Example Program

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch7.b2c" to "C:\…\B2Cv5\ch7.b2c". Then execute the command "build ch7.b2c". Download the ch7.app file to the cybiko.

The example program will take 4 values as input, sum them, and take the average. The average is the sum of a set of numbers divided by the number of items.

```
' chapter 7 example program
' sum and average of 4 grades
' grades are from 0-100
dim grade0  ' we'll discuss the Dim command in the next chapter
dim grade1 ' here are our 4 grades
dim grade2
dim grade3
dim sum    'this variable will store the sum of the 4 grades
dim avg    'this variable will store the average of the 4 grades

' ---INPUT ---
print "Enter grade 0"  'we'll discuss numbering at 0 in the next chapter
input grade0               'get the grades from the student
print "Enter grade 1"
input grade1
print "Enter grade 2"
input grade2
print "Enter grade 3"
input grade3

'--- PROCESS ---
' compute the sum and the average
sum = grade0+grade1+grade2+grade3
avg = sum/4

'--- OUTPUT ---
print "The average of your "
print "4 grades is", avg

print "Press <Enter> to continue"
dim tmp ' a temporary variable
input tmp 'wait for the user to press enter
```

Chapter 7 : Variables and DIM

Variables

As we saw in Chapter 7, variables hold a value. The value can be a number (like 1) or the result of an operation (like a+1). You may think of variable as a shoebox with a label on the outside. Inside the shoebox is a value. On the front of the shoebox is a label with a variable name (like 'a', or 'b' or 'avg').

We also saw that before using a variable we have to declare it. This is done with the 'Dim' statement (short for Dimension). The Dim statement alerts B2C to the fact that we are about to use a variable. The variable is initially set (initialized) to zero.

Data Types

It turns out that B2C has 4 different variable types. These are Int, Char, Long and Double. Currently, Double (or floating point) data types are not supported. Using the Double type will coerce the variable to Int.

'Long' variables are integer variables (no decimal point) containing 4 bytes. They can range from –2,147,483,648 to +2,147,483,647. Because Longs are integer variables, they are fast variables when it comes to computation. But, since they are 4 bytes long, they still take up a lot of Cybiko computing power when processing. Use the Long data type when you need large values but you don't need a decimal point. To declare a Long variable place the words "as long" after the declaration:

```
dim a as long
```

The usual (default) type is called Int (or integer). 'Int' variables are integer variables which means they cannot hold a decimal point and contain only 2 bytes of data. They can range from -32,768 to +32,767. As you will see, the range is much smaller than a Long. And because this is the favorite size of numbers for the Cybiko's internal microprocessor, Int variables perform the fastest. To declare an Int variable place the words "as int" after the declaration:

```
dim a 'defaults to int
dim a as int
```

'Char' variables are also integer variables, but contain only 1 byte of data. Char variables range from -128 to +127. They are not very useful in computations – and since they are smaller than the Cybiko's favorite word size (2 bytes) it actually takes longer to process a Char than an Int. As we'll see shortly, Char variables are usually used in an array as a character string for input and output, rather than as a numeric quantity for computation. To declare a Char variable place the words "as char" after the declaration

```
dim a as char
```

Names of Variables

The name of a variable can be any length with no spaces in it. The first letter must be alphabetic (a-z). Subsequent letters may be alphanumeric (a-z or 0-9). You may also use the underscore (_) character in variable names. As mentioned before, upper and lower case do not matter.

Arrays

An array is a list, or grouping, of variables. If you have a list of something you will want to declare an array of them. Returning to our shoebox example, suppose you have an exam grade. You can mark the outside of a shoebox with the name "grade" and put the exam inside the box. If you ever want to refer to the grade, you find the shoebox marked "grade" and take out the exam and look at the grade you received. This use of a single variable is called a 'scalar'. But you cannot put more than one grade inside the shoebox.

Now, let's assume that you have many students and you want to keep all their grades separated. You might get a set of shoeboxes and line them up in a row. On the front of each box put the student's number (0, 1, 2,

3, etc…). Each student exam goes into the box with that student's number on it. From then on, you access the boxes as "grade[0]", "grade[1]", etc… This is called an array of variables. The part between the brackets is called the subscript.

To create an array of variables, use the Dim statement as before and put the number of elements of the array in brackets after the name. Here is an example of an array of 10 grades:

Dim grades[10] as int

Humans are taught to count starting from one. If I asked you to count to ten, you might reply "1, 2, 3, …10" But computers prefer to count starting from zero. (Technically speaking, computers use an offset and it is convenient for computation and memory layout if the offset starts at zero). So, the first grade in our example is grade[0] (Does this explain why I used grade0 earlier? I hope so!). The next one is grade[1]. And so on up to grade[9]. (Notice that we cannot use grade[10] – since that would be the eleventh grade and we defined 10 integers.).

All data types may be arrays. The Char array is special in that it is treated like a string of text by B2C. When you want to input a name, use the Char array:

```
Dim name[32] as char
Input name
Print "your name is ", name
```

Example Program

DO THIS
Copy the file "c:\…\B2Cv5\tutorial\ch8.b2c" to "C:\…\B2Cv5\ch8.b2c". Then execute the command "build ch8.b2c". Download the ch8.app file to the cybiko.
Here we return to our grade-averaging problem, only this time we use an array of grades. In the next chapter we will see the real power of arrays when we introduce loops.

```
' chapter 8 example program
' sum and average of 4 grades
' grades are from 0-100
dim grade[4]  ' an array of 4 grades
dim sum    'this variable will store the sum of the 4 grades
dim avg    'this variable will store the average of the 4 grades

' ---INPUT ---
print "Enter grade 0"
input grade[0]            'get the grades from the student
print "Enter grade 1"
input grade[1]
print "Enter grade 2"
input grade[2]
print "Enter grade 3"
input grade[3]

'--- PROCESS ---
' compute the sum and the average
sum = grade[0]+grade[1]+grade[2]+grade[3]
avg = sum/4

'--- OUTPUT ---
print "The average of your"
print "4 grades is", avg

print "Press <Enter> to continue"
dim tmp ' a temporary variable
input tmp 'wait for the user to press enter
```

Chapter 8 : Looping : For … Next

The computer is very good at following instructions. Its ability to do what it is told to do, over and over again, is what makes the computer a valuable tool. In this chapter we will learn how to instruct a computer to do the same thing several times.

This ability is called looping. It is called looping because in the early days of computers programmers drew pictures of their programs before they ever wrote code. These pictures were called "flow charts." In a flow chart a statement in a program was shown as a box. The next statement was connected to the first by a line with an arrow on it. If a programmer wanted to repeat a step in the process, they would show it by drawing a line back up to the previous step – forming a loop. (There is also the story of how paper tape with little holes in it was used to instruct the computer. To make the computer repeat instructions over and over, the programmer would connect the end of the tape to the beginning – forming a loop.)

In B2C the loop is implemented by the For command. The For command has three parts: the initialization, the final value, and the (optional) step.

```
for [initialization] to [final value] [step]
for i=0 to 3 step 1
    'statements go here
next
```

The 'initialization' part uses an un-Dim'd variable name (like i) and is initialized to the start value of the loop. The 'to' part declares the last value of the loop and the 'step' declares the amount to increment 'i' inside the loop. If the 'step' part is omitted "step 1" is assumed. The word "next" indicates the end of the loop and tells B2C to increment the variable by the step value and to go back to the top of the loop (the statement just after the for). If the variable has reached the 'final value' then processing continues on the statement after the next.

For readability, the statements inside the for loop are indented 3 or 4 spaces.

It is possible to exit a For loop early by executing the "Exit For" command in the middle of the loop.

Historical Note: The variables i, j, k, l, m, & n are favorite variables for loops. This is for two reasons. First, the single letter makes for easier typing as a subscript to an array variable. Second, in the early days of programming (1960s) there was a language called FORTRAN that had no DIM statement. Instead, all variables were named a-z, and were assumed to be Double (floating point). Except for the variables i-n, which were integers. Why i-n? Because i & n were the first 2 letters of the word "integer".

Infinite Loops

It is possible to get caught in a loop that never exits. This is known as an infinite loop. A for loop with a zero step size would create an infinite loop

```
for i=0 to 0 step 0
  print i
next
```

The example code above will print a screen ful of zeroes. To get out of an infinte loop, press the ESC key.

Example program:

DO THIS
Copy the file "c:\…\B2Cv5\tutorial\ch9.b2c" to "C:\…\B2Cv5\ch9.b2c". Then execute the command "build ch9.b2c". Download the ch9.app file to the cybiko.
With the introduction of loops, our program becomes even simpler. Now we don't have to declare the number of grades up front. It becomes a variable we get from the user at the start.

```
' chapter 9 example program
' sum and average of n grades
' grades are from 0-100
dim sum    'this variable will store the sum of the n grades
dim avg    'this variable will store the average of the n grades
dim n as int   ' the number of grades to average

print "How many grades?"
input n
dim grade[n]  ' an array of n grades

 ' ---INPUT ---
for i=0 to n-1                  'get the inputs
    print "Enter grade ", i    ' notice that we indent in loops
    input grade[i]                 'get the grades from the student
next

'--- PROCESS ---
' compute the sum and the average
for i=0 to n-1
    sum = sum + grade[i]   'notice we accumulate the sum
next
avg = sum/n                'compute the average

'--- OUTPUT ---
print "The average of your"
print n, " grades is", avg

print "Press <Enter> to continue"
dim tmp ' a temporary variable
input tmp 'wait for the user to press enter
```

Chapter 9 : Conditionals : If … Then … Else … End If

Another thing computers are good at is making unbiased decisions. In B2C this is implemented with the If statement. The If statement has a conditional followed by the Then clause:

```
if [conditional] then
  'statements
end if

if a < b then
    print "a is smaller"
end if
```

This conditional has a left side and a right side with a relational operator in between. You can compare two variables, or two expressions. There are 6 relational operations:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal to
<>	not equal to

For example:

```
if a+1 < b*2 then
    'statements
end if
```

There is also an Else statement, which indicates what to do if the conditional is not met:

```
if [conditional] then
    'statements
else
    'else statements
end if
```

```
if a<b then
    print "a is smaller"
else
    print "b is smaller"
end if
```

If you inspect our last example closely – you will see an error in the logic. The error occurs when a is equal to b. In this case our example will print "b is smaller" which is of course incorrect. To correct this we have the Elseif statement. It works just like the If statement and is executed only when the if part is false.

```
if [conditional] then
    'statement
elseif [conditional] then
    'elseif statements
else
    'else statements
endif
```

```
if a<b then
    print "a is smaller"
elseif b<a then
    print "b is smaller"
else
    print "a and b are equal"
end if
```

You may have as many Else if statements as you like, and you may have as many statements in the if/elseif/else clauses as you like. Again. it is good practice to indent the statements to show which statements belong to which if/elseif/else clauses.

Combining Conditionals

The result of the relational operators (<,>,<=,>=,=,<>) is either TRUE or FALSE. This type of value is called Boolean (named for the mathematician George Boole). Booleans are either True or False (one or zero). These types of results can be combined with the Boolean operators AND and OR. You can even negate the conditional with the NOT operator.

The And operator results in a TRUE value if BOTH of the inputs are TRUE (and results in a FALSE otherwise). The Or operator results in a TRUE value if EITHER of the inputs are TRUE. In B2C, a zero value is FALSE and any other value is considered TRUE. NOT converts a TRUE result into a FALSE result.

```
if a<b and b<c then
    print "b is between a and c"
end if

if a>b or a>c then
    print "a is bigger than either b or c"
end if
```

```
a=1
b=0
if a and b then
    print "both a and b are true"
end if

if a or b then
    print "either a or b is true"
end if
```

Example program:

DO THIS
Copy the file "c:\…\B2Cv5\tutorial\ch10.b2c" to "C:\…\B2Cv5\ch10.b2c". Then execute the command "build ch10.b2c". Download the ch10.app file to the cybiko.
Conditionals will allow us to assign a letter grade to the input grades and the average

```
' chapter 10 example program
' sum and average of n grades
' grades are from 0-100
dim sum    'this variable will store the sum of the n grades
dim avg    'this variable will store the average of the n grades
dim n as int   ' the number of grades to average

print "How many grades?"
input n
dim grade[n]  ' an array of n grades

 ' ---INPUT ---
for i=0 to n-1                'get the inputs
    print "Enter grade ", i   ' notice that we indent in loops
    input grade[i]               'get the grades from the student
next

'--- PROCESS ---
' compute the sum and the average
for i=0 to n-1
    if grade[i] >=94 then
       print grade[i], "=A"
    elseif grade[i]>=86 then
       print grade[i],"=B"
    elseif grade[i]>=78 then
       print grade[i], "=C"
    elseif grade[i]>=70 then
       print grade[i], "=D"
    else
       print grade[i], "=F"
    end if
    sum = sum + grade[i]   'notice we accumulate the sum
next
avg = sum/n

'--- OUTPUT ---
print "The average of your",
print n, " grades is ", avg
if avg >=94 then
       print avg, "=A"
elseif avg>=86 then
        print avg,"=B"
elseif avg>=78 then
       print avg, "=C"
elseif avg>=70 then
       print avg, "=D"
else
       print avg, "=F"
end if

print "Press <Enter> to continue"
dim tmp ' a temporary variable
input tmp 'wait for the user to press enter
```

Chapter 10 : While

The While command is another type of looping command. Like the For command, While executes until a condition is met. Here is the form of the While command:

```
while [conditional]
  'things to do
wend
```

```
dim a as int
a=1
while a<10
   print "a=", a
   a=a+1
wend
```

The Wend command ends a while loop.

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch11.b2c" to "C:\…\B2Cv5\ch11.b2c". Then execute the command "build ch11.b2c" Download the ch11.app file to the cybiko

This example program plays "Hi/Low" – a number game where the user has to guess the computer's random number

```
'Hi Low Number Guesser
dim guess as int
dim tries as int
dim number as int
tries=0
number = rnd(100)+1
while(guess <> number)
    print "enter your guess"
    input guess
    if guess < number then
        print guess, " is too low"
        tries=tries+1
    elseif guess > number then
        print guess, " is too high"
        tries=tries+1
    end if
wend
print "You guessed it."
print "The number was ", number
print "It took you ", tries, " tries"
dim a
input a
```

Chapter 11 : Functions & Subroutines

You may have noticed in the last example program that we had to create two sets of If/then/else/endif statements to handle the conversion of a grade into a letter grade. This duplication of code is a bad practice. If there are several of these blocks of code, and we find a bug in one of them, we have to update all of them. For example, what if, in our example program, we decided that the grade cutoffs were 90, 80, 70, and 60? We would have to make the modification in each of the two blocks of If statements. This could lead to extra work, or even errors in our program if we forget to make the same changes in both places.

Fortunately, B2C has a concept called a Subroutine. In the old days, a program was called a routine. A Subroutine therefore is a routine within a routine. You can declare a subroutine with the word Sub:

```
Sub [routine-name] [(var as type, var as type, …)]
   'statements
End Sub
```

The Parameters portion of the Subroutine is a list of variables (declared much in the same way as the Dim statement). These variables are the inputs to the subroutine. Once declared, a subroutine can be called (or invoked) with the Call statement

```
[Call] routine-name [(parameters)]
```

The keyword Call is optional.

Here is an example using our grade-averaging program:

```
Sub letterGrade(grade as int)
    if grade >=94 then
       print grade, "=A"
    elseif grade>=86 then
       print grade,"=B"
    elseif grade>=80 then
       print grade, "=C"
    elseif grade>=74 then
       print grade, "=D"
    else
       print grade, "=F"
    end if
End Sub
```

You may exit a Subroutine early by executing the command Exit Sub.

Another way to solve this problem is with a Function. A function is like a subroutine, but it returns a value. In our example a function could return the letter grade, given the numeric grade. To return a value, you set the name of the function to the value, just like an assignment statement. As with the Subroutine, you may exit a function early with the Exit Function command.

```
Function letterGrade(grade as int) as char
    if grade >=94 then
       letterGrade = 65 ‘ascii characters but we didn’t teach it
    elseif grade>=88 then
       letterGrade = 66
    elseif grade>=80 then
       letterGrade = 67
    elseif grade>=74 then
       letterGrade = 68
    else
       letterGrade = 70
    end if
End Function
```

Example program:

DO THIS
Copy the file "c:\…\B2Cv5\tutorial\ch12.b2c" to "C:\…\B2Cv5\ch12.b2c". Then execute the command "build ch12.bld". Download the ch12.app file to the cybiko Subroutines allow us to simplify our program by taking similar sections of code and condensing them into one subroutine.

```
' chapter 11 example program
' sum and average of n grades
' grades are from 0-100
Sub letterGrade(grade as double)
    if grade >=94 then
       print grade, "=A"
    elseif grade>=86 then
        print grade,"=B"
    elseif grade>=80 then
       print grade, "=C"
    elseif grade>=74 then
       print grade, "=D"
    else
       print grade, "=F"
    end if
End Sub

dim sum    'this variable will store the sum of the n grades
dim avg    'this variable will store the average of the n grades
dim n as int   ' the number of grades to average

print "How many grades?"
input n
dim grade[n]  ' an array of n grades

 ' ---INPUT ---
for i=0 to n-1                'get the inputs
    print "Enter grade ", i   ' notice that we indent in loops
    input grade[i]               'get the grades from the student
next

'--- PROCESS ---
' compute the sum and the average
for i=0 to n-1
    print "Grade ", i, ": ", grade[i]
    call letterGrade(grade[i])
    sum = sum + grade[i]   'notice we accumulate the sum
next
avg = sum/n

'--- OUTPUT ---
print "The average of your"
print n, " grades is ", avg
call letterGrade(avg)

print "Press <Enter> to continue"
dim tmp ' a temporary variable
input tmp 'wait for the user to press enter
```

Chapter 12 : Math Operations

B2C supports a host of math Operations. The ordinary operations exist (+, -, *, /,MOD) with which you can do math calculations. The Order of Operations applies (multiplication and division first, then addition and subtraction) and parentheses are used to change the order of operations. (a=(a+b)*c).

B2C currently supports only integer arithmetic.

```
dim a as int
a = 5 + 1
print "the answer is ", a
> the answer is 6
```

Random Numbers

Random numbers are numbers whose value is unpredictable. You can get a random number with the Rnd() function:

Rnd(x)

Where 'x' is any number. Rnd(x) will return a number from 0 to 'x' inclusive.

x=rnd(5)+1 'return a number from 1 to 6, like a 6-sided die

Chapter 13 : Simple Graphics

The Cybiko is a very capable graphics machine. In this section we cover the non-bitmapped graphics commands. B2C has 5 basic graphic commands with which to draw on the screen.

Cls – Clear screen

Cls stands for Clear Screen. It will turn all pixels on the display to the background color (usually white). It will also erase all text on the screen. Cls takes no parameters

```
Cls
```

Paper – set the color of the background

B2C has a concept of a foreground color and a background color. The background color is set with the Paper command. Whenever you do a Cls – the entire display is set to the color defined by the Paper command. There are 4 colors to chose from. Color 0 is white. Color 1 is light grey. Color 2 is dark grey. Color 3 is black. There are constants defined for your use:

0 – White
1 – LtGrey
2 - DkGrey
3 - Black

```
Paper Black ' set the background color to black
Cls ' color the screen with the background color
```

Ink – set the color of the foreground

B2C's foreground color is used to draw text (Print) and lines (Line). There are constants defined for your use:

0 – White
1 – LtGrey
2 - DkGrey
3 - Black

```
Paper White ' set the background to White
Ink Black ' set the foreground to black
Cls ' clear the background
line -80, -50, 80, 50 ' draw a diagonal line
```

Line – draw a line on the screen

The B2C display area is an X/Y coordinate system. You place a line on the screen by telling B2C to position the cursor at a horizontal (X) coordinate and a vertical (Y) coordinate. The valid values for the screen coordinates are –80 to 79 in the X (horizontal) direction, and –50 to 49 in the Y (vertical) direction. You may specify values as large as +/- 20,000 in either direction, but only the pixels, which are in the screen coordinates (-80 to 79 and –50 to 49), will be displayed.

```
Paper White ' set the background to White
Ink Black ' set the foreground to black
Cls ' clear the background
line -80, -50, 80, 50 ' draw a diagonal line
```

It is possible to use a different coordinate system. The so-called 'C' coordinate system runs from 0 to 159 in the X direction and from 0 to 99 in the Y direction. You can switch to this system by placing

OPTION C_COORDS

at the top of your program.

Point – draw a single pixel on the screen

Draw a single pixel at the x/y coordinates (see Line, above).

```
Paper White ' set the background to White
Ink Black ' set the foreground to black
Cls ' clear the background
Point 0, 0 ' draw a dot in the center
```

Example Program

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch13.b2c" to "C:\…\B2Cv5\ch13.b2c". Then execute the command "build ch13.bld". Download the ch13.app file to the cybiko.

This program is like the old Microsoft Windows screen saver. A set of bouncing lines chase after each other.

```
'chapter 13
'qix line drawing
'requires B2C-2
'see below for B2C-1

dim n as int
print "Number of lines"
input n

dim n1 as int
dim n2 as int
n1=n-1
n2=n1-1

dim x0[n] as int
dim y0[n] as int
dim x1[n] as int
dim y1[n] as int
dim dx0 as int
dim dy0 as int
dim dx1 as int
dim dy1 as int
dim seed as int

sub newdirection(j as int)
  if (x0[j] < -80 or x0[j] > 80) then
    dx0 = -dx0
  end if
  if (x1[j] < -80 or x1[j] > 80) then
    dx1 = -dx1
  end if
  if (y0[j] < -43 or y0[j] > 43) then
    dy0 = -dy0
  end if
  if (y1[j] < -80 or y1[j] > 80) then
    dy1 = -dy1
  end if
end sub

' for B2C1 do the following
' dx0=-3
' dx1=-1
' dy0=1
' dy1=2
'and remove the while loops...

' select random motion offsets
while dx0=0
  dx0 = 4-rnd(9)
wend
while dx1=0
  dx1 = 4-rnd(9)
wend
while dy0=0
  dy0 = 4-rnd(9)
wend
while dy1=0
  dy1 = 4-rnd(9)
wend

'select initial random line
x0[0] = rnd(80)
x1[0] = rnd(80)
y0[0] = rnd(43)
y1[0] = rnd(43)
line x0[0], y0[0], x1[0], y1[0]

cls
```

```
'fill up queue with trailing lines and print them
for i=1 to n-1
  x0[i] = x0[i-1] + dx0
  x1[i] = x0[i-1] + dx1
  y0[i] = y0[i-1] + dy0
  y1[i] = y1[i-1] + dy1
  line x0[i], y0[i], x1[i], y1[i]
  newdirection(i)
next

for i=0 to 1 step 0
  ink 0
  line x0[0], y0[0], x1[0], y1[0]
  ink 3
  for j=1 to n-1
    x0[j-1] = x0[j]
    x1[j-1] = x1[j]
    y0[j-1] = y0[j]
    y1[j-1] = y1[j]
  next
  x0[n1] = x0[n2] + dx0
  y0[n1] = y0[n2] + dy0
  x1[n1] = x1[n2] + dx1
  y1[n1] = y1[n2] + dy1
  line x0[n1], y0[n1], x1[n1], y1[n1]
  newdirection(n1)
next
```

Chapter 14 : Printxy

Printxy is like a combination of the Point function and the Print function. Printxy allows you to position text anywhere on the Cybiko screen. The format of the Printxy command is:

```
printxy x,y, value [, value…]
```

The text string is composed of the values of the variables after the Y coordinate. The text string is printed at the coordinates X, Y. The values will be painted with the X/Y coordinate in the upper left corner of the text. The values can be any collection of comma-separated variables or literals – just as in Print.

Our Example Program will print your name on the display and bounce it off the edges of the screen. Try using shorter names to see how the performance improves.

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch14.b2c" to "C:\…\B2Cv5\ch14.b2c". Then execute the command "build ch14.b2c". Download the ch14.app file to the cybiko.

```
dim name[32] as char'your name
dim x as int        'x coordinate of your name
dim y as int        'y coordinate of your name
dim dx as int      'direction the name moves in x coord
dim dy as int      'direction the name moves in y coord
input "Enter your name ", name
x = 80-rnd(160)  'random x starting point
y = 43-rnd(86)     'random y starting point
dx = -1  ' moving to the left
dy = -1  'moving up
while 1  'do forever or until ESC is pressed
    cls                          'clear the screen
    printxy x,y, name   'print the name onscreen
    x=x+dx   'move x dir
    y=y+dy   'move y dir
    if x>80 then
      dx=-dx    ' bounce off right side
    end if
    if x<-80 then
      dx=-dx   ' bounce off left side
    end if
    if y>30 then
      dy=-dy   'bounce off bottom
    end if
    if y<-43 then
      dy=-dy  'bounce off top
    end if
wend
```

Chapter 15 : Miscellaneous

Beep

B2C includes a simple command for making sound. It is different from the beep command in CyBasic2. The Beep command causes the speaker to make a single tone. Beep takes values from 1 (low-pitch) to 63 (high-pitch). Beep 0 will result in silence.

Wait

There are 2 ways to create a delay in B2C. One way is to write a For loop which does nothing:

```
for i=0 to 100
next i
```

This is not recommended. It is sloppy coding and will be unpredictable in the event that Cybiko, Inc releases a newer, faster Cybiko. The alternative is the Wait command. The parameter is a number of tenths of seconds to wait. So, to wait one second:

```
wait 10
```

Key

The Key command has two modes. In the first, it acts like a variable.

x = key

It returns the value of the last key pressed on the keypad. This is very useful for writing interactive games because it does not stop the action like the Input statement does. At the end of this chapter is a table of all the values the Key variable can return. To use the table, find the key you want to check against and add the values in the corresponding row and column. For example, the letter 'a' is 90+7=97.

The other format for the key command is as a function to check a single key:

x = key(#KEY_ENTER)

In this mode you specify the key you want to check and key() returns True or False. Here is a listing of some of the constants you may use (note, these must be in uppercase)…

#KEY_A - #KEY_Z
#KEY_0 - #KEY_9
#KEY_UP, #KEY_DOWN, #KEY_LEFT, #KEY_RIGHT
#KEY_ENTER, #KEY_SPACE, #KEY_INS, #KEY_DEL

Key Values	0	1	2	3	4	5	6	7	8	9
30		space								quote
40					comma	dash	period	/	0	1
50	2	3	4	5	6	7	8	9		semi-colon
60		=								
90		(	\	)			back-quote	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z							
260					left-arrow	up-arrow	right-arrow	down-arrow	ins	del

Key Values	0	1	2	3	4	5	6	7	8	9
270	tab	select	enter	bspc		shift	fn			

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch16.b2c" to "C:\…\B2Cv5\ch16.b2c". Then execute the command "build ch16.b2c" Download the ch16.app file to the cybiko.

Here is an example of a program that moves a dot left or right:

```
dim x as int    'horizontal component of the dot
dim y as int    'vertical component of the dot
dim z as int  'the key pressed
x=0    'initial x position of the dot
y=0    'initial y position of the dot
cls     'clear screen
point x,y  'show the dot
for i=0 to 1 step 0  'loop forever
z = key              'save off the key because it changes
  if z = 264 then    'left key
    ink 0
    point x,y
    ink 3
    x=x-1            'move left
    point x,y        'print new dot
  elseif z = 266 then' right key
    ink 0
    point x,y
    ink 3
    x=x+1            ' move right
    point x,y        ' print new dot
  end if
next 'end of forever loop
```

Chapter 16 : String Manipulation (Right/Mid)

The B2C language offers no way to assign one string to another:

```
dim a[32] as char
dim b[32] as char
:
a = b  'illegal operation
```

Instead there are 2 statements for copying strings. These are the Mid statement (used for copying the middle of one string to another) and the Right statement (used for copying the right side of a string). The Mid and Right statements takes the form:

```
Mid destination, source, start [, length]   'the length is optional
Right destination, source, length
```

For the Mid statement, the 'destination' is a dimensioned string where the result will be stored. The contents of the original string will be discarded. The 'source' is a dimensioned string that holds data to be copied to the destination. The 'start' is a numerical value (variable, constant, or expression) indicating the first character from the 'source' to copy (remember, in B2C we count starting at zero). The optional length indicates how many bytes to copy. If the length is omitted, or if it is larger than the number of characters in the source array, the entire source array is copied.

The Right statement has similar parameters. The destination is the dimensioned string to copy into, and the source is the string to copy from. The length parameter is required and indicates how many letters from the RIGHT of the source string to copy into the destination.

In our example program we will learn to speak pig latin. It is not perfect. Can you think of ways to improve it?

Example Program

DO THIS

Copy the file "c:\…\B2Cv5\tutorial\ch18.b2c" to "C:\…\B2Cv5\ch18.b2c". Then execute the command "build ch18.b2c" Download the ch18.app file to the cybiko

```
'ch 18
dim word[32] as char     ' the word to convert
dim ordway[32] as char ' the resultant word
dim len as int                ' the length of ordway

input "enter your word ", word  'input
mid ordway, word, 1                'copy the word but not the 1st char

len = 0
for i=0 to 31                              'get the ordway length so we
can add 'ay'
    if ordway[i] = 0 and len = 0 then
      len = i
    end if
next
'print len
ordway[len] = word[0]    ' append the first letter of word to the end of
ordway
ordway[len+1] = 97        ' append 'a'
ordway[len+2] = 121      ' append 'y'
ordway[len+3] = 0          ' null terminator, very important

print word, "=", ordway   ' print results
```

```
input word ' wait for Enter
```

Chapter 17 : File I/O

Files on the Cybiko are stored in the non-volatile memory. They are like files on the PC in that they have filenames and they hold data. They even have attributes (like file size), but these attributes are not accessible from B2C.

Open a file

Before a file can be accessed, it must be opened. In B2C, opening a file is accomplished with the Open command. The Open command has three parts – the pathname, the mode, and the filenumber. The mode is optional.

```
open pathname [for mode] as filenumber
```

The pathname is the name of the file. This can be either a literal string ("filename.dat") or a variable which has been Dim'd previously. The mode is optional and is one of Read, Write, or Append. If left blank, the mode defaults to Read. When a file is opened for Read access the program may only read data from the file, no writing is permitted. Likewise, when a file is opened for Write, no data may be read from the file, only written. If the file already exists, the data in the file is erased. When a file is opened for Append, it is opened for write, but if the file exists, instead of destroying the data, the file pointer is positioned at the end of the file and writing begins there. Finally, the filenumber is a number from 0 to 7. It is used to identify the file for the rest of the program.

```
Dim fname[32] as char

print "Enter fname"
input fname
open fname for read as 1
```

It is possible to open the same file for Read in more than one place in the program. But for Write and Append modes you must first close the file before reopening it.

Close a file

When you are done with a file, you must close it. The Close command takes as its only parameter the filenumber…

```
open "filename.dat" as 1
 'do some stuff
close 1
```

Writing to a File : Put

The Put statement writes data from a variable to a file. It has three parts: filenumber, bytepos, and variable

```
Put filenumber, [bytepos][, variable]
```

The Put command keeps the concept of the file position. Each time data is written to the file the file position is incremented by the size of the variable in bytes. In this way, you can accurately control the data being written to the file. The first byte in the file is byte 0, the next is byte 1, etc…

```
dim foo as int
open "filename.dat" for write as 1
put 1, 0, foo  ' write two bytes at the beginning of the file
put 1, 100, foo ' write the same bytes at the 100th byte of the file
```

Leaving out the variable name positions the file pointer, but does not write

```
dim foo as int
open "filename.dat" for write as 1
put 1, 0  ' position file pointer to the beginning of the file
```

To write to the current file pointer position without specifying the value, leave out the bytepos (but remember to include the delimiting commas). Unfortunately string variables and other arrays cannot be written with the Put command. You must create a loop and write each element individually.

```
dim a as double
open "filename.dat for write as 1
input a
put 1,,a ' write 8 bytes to the current location
```

Reading from a file : Get

The Get statement reads data from a file into a variable. It has three parts: filenumber, bytepos, and variable

```
Get filenumber, [bytepos][, variable]
```

The Get command keeps the concept of the file position. Each time data is read from the file the file position is incremented by the size of the variable in bytes. In this way, you can accurately control the data being read from the file. The first byte in the file is byte 0, the next is byte 1, etc…

```
dim foo as int
open "filename.dat" for read as 1
get 1, 0, foo  ' read two bytes from the beginning of the file
get 1, 100, foo ' read two different bytes from the 100th byte of the file
```

Leaving out the variable name positions the file pointer, but does not read

```
dim foo as int
open "filename.dat" for read as 1
get 1, 0  ' position file pointer to the beginning of the file
```

To read from the current file pointer position without specifying the value, leave out the bytepos (but remember to include the delimiting commas).

```
dim a as double
open "filename.dat for read as 1
get 1,,a ' read 8 bytes from the current location
print a
```

Printing to the file : Print #

Printing to a file is possible with the print statement, which you are already familiar with. Just add a "#n" where n is the filenumber. This is an easy way to create text files. *Strangely, there is no corresponding Input # command for reading data.*

```
dim a[10] as char
input "your name", a
open "filename.txt" for write as 1
print #1, a
close 1
```

Chapter 18 : TARDIS: All Text Role Playing Game

The programs that are the most fun to write are games. Graphic games (like Super Mario Brothers) are complex programs requiring bitmapped graphics, which is covered in the guide "BitMap.doc". Other graphic games (like tic-tac-toe) are possible, but text-based games are a good starting point.

B2C is well suited to writing all-text role-playing games (RPGs). A role-playing game is one where you act as a character in a world created by the computer. You can interact with objects in that world and move from place to place. I will present a simple role-playing game using a generic approach. There are many ways to write RPGs, so you should experiment and find the one that makes most sense to you.

The first step in creating a RPG is deciding what your universe will be like. Is your setting in outer space? Or the deep sea? What is your experience level? Are you an expert spy? Or a novice cowboy? What is your goal in the adventure? Are you trying to get to the end of a maze? Or are you attempting to collect all the Dragon Balls? Regardless of the subject matter, a good RPG tells a story, and the more detailed your descriptions of the adventure, the more interested your user will be and the more exciting your game will be.

Once you have decided on a universe, make a map of the universe. Make a box for each room and a line with an arrow showing how to go from one room to the next. Keep it down to 5-10. Make a description for each room. Make a list of objects and decide where in universe these objects reside. Make a description for each object. Make a list of commands, like "north" to go north and "eat" to eat food.

The next step is a technical one. How do you get your inputs from the user? Will you list their options and have them input a number selecting what to do? Or will you have them input text strings and discover what commands work best.

The TARDIS Adventure

This RPG is located in the TARDIS. It stands for Time and Relative Dimensions in Space. The RPG is based on the British TV Series "Dr. Who". The TARDIS is "dimensionally transcendental" which simply means that it is larger on the inside than on the outside. While the TARDIS is about the size of a telephone booth, it has many chambers inside. The goal of our game will be to enter the TARDIS, find the sonic screwdriver, and exit.

In this game each room is a separate Function. When you enter the function a description of the room is printed. When the function exits, it returns the number of the next room to go to. This offers our program modularity. Rather than a linear adventure that runs from the top of the program to the bottom, the user can wander from room to room and even get caught in a trap between rooms.

There are several tricks in this program. One trick is that the return value from each room is an integer describing the next room to visit. Using this method you can rewrite the adventure by changing descriptions and the return rooms. Another trick is "parsing". I create a variable with the value of the lettername of that variable. (eg: a=97). When I parse I check each letter in the "verb" against the variable. So, for "eat" I check verb[0]=e and verb[1]=a and verb[2]=t as well as verb[3]=0 (null terminator).

I allow you to carry 2 objects (one in each hand). And you can drop only 1 object in each room. The only commands are :

Command	Meaning	Description
n	go north	
e	go east	
s	go south	
w	go west	
I	Inventory	Lists all the objects you are carrying
eat	eat something	increase your energy
take	take something	move an object from right hand to left hand and pick up something in your right hand. do nothing if hands full
drop	drop something	drop something from your left hand move other object from right hand to left hand. cannot drop if room is full

DO THIS
Execute the command "build tardis.bld". Download the tardis.app file to the cybiko.

Chapter 19: Bit Mapped Graphics

B2C Version 2c introduced Bitmapped Graphics. Four commands (Sprite, Move, Collision, and Redraw) make for a complete and simple interface to write bitmapped and animated applications. Other commands introduced with Version 2c are Vibrate and Music to make for a well-rounded gaming experience. Finally, the key() function was extended to allow checking for a single keystroke, thus improving performance.

The program "gfx.app" in the ***Gfx*** directory demonstrates these capabilities. It shows 10 lemmings walking around the screen and bouncing off the walls. One lemming is upside down and when it collides with another lemming the Cybiko vibrates. Take a minute to "build" this app and download it to your Cybiko to test.

Getting Started

The heart of any video game is the animated character called a "Sprite". (The word Sprite literally means ghost, spirit, fairy or elf). To create a sprite, create a bitmap using any draw tool (Paint.exe in MS Windows works fine). Your sprite bitmap can be any size, but typically they are 8x8 pixels up to 20x20. (The X and Y dimensions don't have to match). You should use grayscale colors of Black, Light Gray, Dark Gray, and White to draw your sprite.

Sprite Command

The Sprite command in B2C allows you to load a sprite into the program from the .app file. The format of the sprite command is:

```
Sprite n, "filename.pic"
```

The 'n' in this case is the sprite number. There are 32 sprites in B2C. The sprite number 0 is usually used for the background sprite while sprites 1-31 are used for characters. In our example you will need to load the sprite "sprite.pic" into sprite number 1…

```
Sprite 1,"sprite.pic"
```

This merely loads the sprite into memory. To position it on screen you need the Move command and to display it you need the Redraw command.

Move Command

The Move command positions your sprite on the screen. It is a complicated command - only the simple interface is presented in this section, a later section will detail the more advanced features.

```
Move n,x,y
```

The 'n' is the sprite number to move and the x,y are coordinates in the typical CyBasic coordinate system (-80 to 79 in the x direction and -50 to 49 in the y direction). Suppose we wanted to move the sprite to the middle of the screen, we would do this…

```
Move 1,0,0
```

Redraw

Nothing is shown on the screen until you execute the Redraw command. This command clears the display and redraws each sprite in its new position.

```
Redraw
```

Sprite Example Program 1 : sprite1.app

Here is a sample sprite program which will load a sprite and move it around the screen based on the arrow keys. This complete application can be found in the "sprite1" directory:

```
Dim x as int                        'x coordinate
Dim y as int                        'y coordinate
X=0
Y=0
Sprite 1, "sprite.pic"              'load the sprite
While(TRUE)                            'do forever
       If key(#KEY_LEFT) then x=x-1       'move left
       If key(#KEY_RIGHT) then x=x+1       'move right
       If key(#KEY_UP) then y=y+1        'move up
       If key(#KEY_DOWN) then y=y-1       'move down
       Move 1, x, y
       Redraw
Wend
```

Changing the sprite1.bld file

In order to get your sprite into the sprite1.app program, you need to update the sprite1.bld file. Add the lines

[pic=sprite.pic]
sprite01.bmp

You can now "build" your program as usual and download the result to your cybiko.

Adding Animation

Our example program is fun, but has some obvious problems. Firstly, the image is static, it doesn't look like the lemming is really walking because his feet don't move. Secondly, when the image is moving to the left it is facing to the right. Both of these problems will be addressed in the next section.

Animation is what makes programming video games fun. To create an animation, you must create a set of bitmaps which are each slightly different from the last. It is also best if the last image leads to the first image naturally. It makes sense to name these bitmaps similarly, like Sprite01.bmp, Sprite02.bmp, etc… The example program "Sprite2" can be found in the "sprite2" directory. In our example we use 5 bitmaps.

Once you have created your animation sequence, you need to store it in the sprite2.bld file like this:

[pic=sprite.pic]
sprite01.bmp
sprite02.bmp
sprite03.bmp
sprite04.bmp
sprite05.bmp

Move: Revisited

The Move command has special parameters specifically for animation. The format for the animated move command is:

Move n, x, y, z, mode

This is identical to the earlier move command. 'n' is the sprite number and x,y are the coordinates. 'z' is the bitmap number to display when the sprite moves and mode determines the direction the sprite faces. The

mode is 0 for normal, FLIP_X for a flip in the x direction, FLIP_Y for a flip in the y direction, and FLIP_X+FLIP_Y for a flip in both directions.

Sprite Example Program 2 : sprite2.app

Here is a sample sprite program which will load a sprite and move it around the screen based on the arrow keys. It also includes animation. This complete application can be found in the "sprite2" directory:

```
Dim x as int                        'x coordinate
Dim y as int                        'y coordinate
Dim z as int                        'bitmap number
Dim mode as int                     'mode
x=0
y=0
z=0
mode=0
Sprite 1, "sprite.pic"              'load the sprites
While(TRUE)                            'do forever
       If key(#KEY_LEFT) then
              x=x-1                    'move left
              mode=FLIP_X              'face left
              z=z+1                    'move feet
              if z>=5 then z=0         'cycle animation
              end if
       If key(#KEY_RIGHT) then
              x=x+1                    'move right
              mode=0                   'face right
              z=z+1                    'move feet
              if z>=5 then z=0         'cycle animation
              end if
       If key(#KEY_UP) then y=y+1         'move up
       If key(#KEY_DOWN) then y=y-1         'move down
       Move 1, x, y, z, mode          'move and animate sprite
       Redraw
Wend
```

Collisions

A Collision is when two sprites intersect. In B2C this intersection is computed using the Bounding Box (rectangle) of the sprite, not the pixels themselves. The function to detect a collision is

Collision(a,b)

Where 'a' and 'b' are two different sprite numbers.

Collisions are useful for determining if a bullet intersects a character, or a punch intersects another sprite.

Music

Music is added using Cybiko's '.mus' file format. This file can be created using the "Converter.exe" program which is part of "Cyberload". The Converter program will read a .midi file and convert it to .mus. As with the sprite file, you must update the '.bld' file to include the filename.

[music]
sprite.mus

The command to load music into a B2C application is

Music Background, "filename.mus"
Or
Music Foreground, "filename.mus"

As with the Sprite command, this only loads the music file into memory. To play the music, you must issue the following command:

Music Background, Play
Or
Music Foreground, Play

Music played in the Background will play over and over again until you stop it. Music played in the foreground will play only once. To stop the music from playing issue the following command:

Music Background, Stop
Or
Music Foreground, Stop

Vibrate

The Vibrate command turns on the Cybiko's vibration device. Here is the format:

Vibrate n

Where 'n' is from 0-255. 0 stops vibration, 128 is medium vibration and 255 is maximum vibration.

Sprite Example Program 3 : sprite3.app

Here is a sample sprite program which will load a sprite and move it around the screen based on the arrow keys. It also includes animation, collision detection, music and vibration:

```
Dim x as int                        'x coordinate
Dim y as int                        'y coordinate
Dim z as int                        'bitmap number
Dim mode as int                     'mode
x=-80
y=49
z=0
mode=0
print "Loading..."
Sprite 1, "sprite.pic"              'load the sprites
Print "Still Loading..."
sprite 2, "root.ico"             'the b2c icon
move 2, -24,24                    'put b2c in the middle
print "Last time Loading..."
Music Background, "sprite.mus"
Music Background, Play
ink WHITE
print "lets go!"
While(TRUE)                             'do forever
        If key(#KEY_LEFT) then
                x=x-1                   'move left
                mode=FLIP_X             'face left
                z=z+1                   'move feet
                if z>=5 then z=0        'cycle animation
        end if
        If key(#KEY_RIGHT) then
                x=x+1                   'move right
                mode=0                  'face right
                z=z+1                   'move feet
                if z>=5 then z=0        'cycle animation
        end if
        If key(#KEY_UP) then y=y+1        'move up
        If key(#KEY_DOWN) then y=y-1        'move down
        Move 1, x, y, z, mode         'move and animate sprite
        vibrate 0
        if collision(1,2) then vibrate 128
        Redraw
Wend
```

Final Thoughts

Because sprites are drawn in sprite-number-order, sprite number 31 appears to be 'on top'. In our last example (sprite3.app) the lemming appears to walk behind the B2C icon because the lemming is drawn first and the B2C icon is drawn second.

Chapter 20: Performance Enhancements

Before reading this guide you should already have read the "Quick Start Guide" (docs/_QuickStart.doc), the "B2C Tutorial" (tutorials/b2c.doc), and the "Bitmapped Graphics Guide" (docs/Bitmaps.doc). These documents are the foundation for all B2C programs. You should also be very familiar with the "Language Reference Guide" (docs/LanguageReference.doc).

There are only a few "tricks" to improving performance in a B2C application. They are
1. KEY(keynumber)
2. OPTION ESCAPE OFF
3. OPTION SHOW OFF

KEY(keynumber)

If you are a seasoned (or even a beginner) CyBasic programmer, then you may be accustomed to checking for a keystroke like this:

```
If key = 267 then
   Print "You hit Enter"
End if
```

This is fine if you are just checking a key once in a while, but for most gaming applications you may want to check for a key in a loop, like this:

```
While key <> #KEY_ENTER
  ' wait for the enter key
wend
```

The problem with this loop is that the "key" function calls the cWinApp_get_message() function. This function never takes less than 1/10 of a second to execute. Which means you will never get more than 10 frames per second, and probably less. To gain maximum performance, you must use the key(n) function (now that's the word *key* with a number in parenthesis). It takes as a parameter a key to sample. If the key is pressed down – it returns TRUE (non-zero). And if the key is not pressed, it returns FALSE (0). Try this, instead…

```
While key(#KEY_ENTER)=FALSE  ' key is not depressed
    ' wait for the enter key
wend
```

But wait, there's more! Another problem people fall into is that they call the key (no parens) function too often. Like if you are writing an editor and you want to know what key is currently depressed. The code may look like this:

```
If key = #KEY_LEFT then x=x-1
If key = #KEY_RIGHT then x=x+1
If key = #KEY_UP then y=y+1
If key = #KEY_DOWN then y=y-1
```

In this case you can expect major performance problems and possibly even errors (since you are throwing away keystrokes with each call to key). This is a much better solution…

```
DIM keystroke as char

Keystroke = key  'called just once
If keystroke = #KEY_LEFT then x=x-1
If keystroke = #KEY_RIGHT then x=x+1
If keystroke = #KEY_UP then y=y+1
```

```
If keystroke = #KEY_DOWN then y=y-1
```

You can even improve upon this using the key(n) function as follows:

```
If key(#KEY_LEFT) then x=x-1
If key(#KEY_RIGHT) then x=x+1
If key(#KEY_UP) then y=y+1
If key(#KEY_DOWN) then y=y-1
```

OPTION ESCAPE OFF

The B2C compiler is shipped with certain options in effect. These options are designed to make programming for the novice much easier. One of these features is ESC key detection. At the top of each WHILE or FOR loop an inline call is made to the C function "escape(0)". This function detects if the ESC key has been pressed, and if it has will display the standard "ESC Pressed -Quit?" dialog. Unfortunately, this function takes about 1/10 of a second to execute. For many programs this is not a problem, but for high performance graphics applications, it's much too slow. This problem can be compounded by nested loops which call "escape(0)" not just 2x as often but as much as x^2.

The fix is to declare "OPTION ESCAPE OFF" at the top of the program. Since this removes escape processing from your program, it will run much faster. However, it also means that your user has no easy way to exit from your program. And to make matters worse, you cannot use KEY(#KEY_ESC) to detect the escape key (CyOS does not allow it for unknown reasons). The only recourse is to use some other key as the exit key. Many people use the SPACE key.

```
While KEY(#KEY_SPACE)=FALSE
      If key(#KEY_LEFT) then x=x-1
      If key(#KEY_RIGHT) then x=x+1
      If key(#KEY_UP) then y=y+1
      If key(#KEY_DOWN) then y=y-1
      Move 1,x,y
      Redraw
Wend
```

OPTION SHOW OFF

Normally, when we use any of the graphic functions like printxy, or redraw, or line, the result of the command is immediately apparent on the screen. That is because an internal drawing function called "Show" is executed right away. You see, all graphics commands are actually drawn on a backing display called a buffer. The buffer is drawn on until a "Show" is performed. Then it becomes visible to the user. This technique of buffering graphics is called "double-buffering".

Because B2C was created for the first time programmer, this Show was performed after each graphics command. The problem is it's very inefficient. It makes more sense to draw everything in the buffer then do a Show at the end (rather than a Show after every command). This is why so many B2C programs are both slow and display lots of flicker.

To save memory, and have more control over how things are drawn you need to specify "OPTION SHOW OFF" which turns off the "Show" after each command. Since you now have more power over the graphics engine, you also have more responsibility. You must call a "Redraw Show" when you are done drawing the display.

Another benefit of "OPTION SHOW OFF" is that you can reuse the same sprite over and over again. For example, in the "Playfield Graphics Tutorial" (in the playfield directory) there are 19 "money" icons. It would be much more practical to load that icon once and use it over and over again.

To use the same sprite over and over again, we have to first load the sprite into memory. Then we move the sprite to the first location, call the redraw function for that sprite (and that sprite only). Then move the

sprite to the next location and redraw it. And so on. Each time we redraw the sprite it acts like a "stamp" imprinting an image of the sprite on the buffered display. Finally, when all the sprites have been imprinted, we call "Redraw Show" and the buffer is copied to the main display for the user to see. Here is a sample :

```
Sprite 1, "money.pic"
CLS
Move 1, 0, 0 'upper left of screen
Redraw 1 ' imprint the money on the buffered display
Move 1, 50, 50 'middle of screen
Redraw 1 'imprint the money on the buffered display
Move 1, 100, 75 'lower right of screen
Redraw 1 'imprint the money on the buffered display
Redraw Show ' show the buffer to the user "Show me the Money!"
```

For more information, check out Part 7 of the "Playfield Graphics Tutorial" (playfield/part7/playfield7.doc)

Chapter 21: 3D Graphics

B2C Version 3d introduced 3Dimensional Graphics. New commands (3dRoom, 3dWall, 3dSprite, 3dMove, 3dRMove, 3dCollision, 3dGet and 3dRedraw) make for a complete and simple interface to write 3Dimensional and animated applications.

The program "3d.app" in the ***3d*** directory demonstrates these capabilities. It is a duplicate of Cybiko's own 3d demo with a twirling Cybiko logo. It allows you to maneuver within a 1000x1000 pixel square room. If you bump into a wall you are stopped. If you walk into the Cybiko logo the Cybiko will vibrate.

Getting Started

The basis for any 3d program is the 3dRoom. By default there are 8 rooms (numbered 0-7). You select the current room with the 3dRoom command:

3dRoom n

The default 3dRoom is room 0. From here, you should define the walls within the room. The command to define a wall is the 3dWall command:

3dWall x0,y0,x1,y1,"filename.tex"

The parameters x0, y0, x1, y1 define a line along which the wall will exist. The "filename.tex" is a texture file which is either a 32x32 or 32x16 bitmap. When the wall is rendered (drawn) the texture file will be drawn over and over again to fill in the blanks along the wall. You may specify as many walls as you like.

The texture files can be created using the bmp2spr program (supplied with the Cybiko SDK). The wall texture may be either 32x32 or 32x16 pixels. The b2cbuild program will perform this translation automatically for you. Add the following lines to your .bld file to convert wall.bmp into wall.tex:

[3dtex]
wall.bmp

Camera

The Camera is a figurative object which defines how the room will be rendered. You can position the camera with the 3dMove command:

3dMove Camera, x, y, dir

Positioning the camera tells the Cybiko how to render the scene. The walls are drawn in a perspective from the camera's x,y position as though it were pointing in a direction dir degrees from the center. The redrawing is accomplished with the 3dRedraw command:

3dRedraw

Sprite Command

The 3dSprite command in B2C allows you to load a sprite into the program from the .app file. The format of the 3dSprite command is:

3dSprite n, "filename.spr"

The 'n' in this case is the sprite number. There are 7 3dSprites per room in B2C. The sprite number 0 is used for the Camera. In our example you will need to load the sprite "column.spr" into sprite number 1…

```
3dSprite 1,"column.spr"
```

This merely loads the sprite into memory. To position it on screen you need the 3dMove command and to display it you need the 3dRedraw command.

The sprite files can be created using the bmp2spr program (supplied with the Cybiko SDK). Sprites must be 32x32 pixels. You may have multiple images in a single sprite (just as with 2d Sprites). The b2cbuild program will perform this translation automatically for you. Add the following lines to your .bld file to convert sprite_n.bmp into sprite.spr:

```
[3dsprite=sprite.spr]
sprite_0.bmp
sprite_1.bmp
sprite_2.bmp
sprite_3.bmp
sprite_4.bmp
```

Move Command

The 3dMove command positions your sprite on the screen.

```
3dMove n,x,y,z
```

The 'n' is the sprite number to move and the x,y are coordinates in the typical CyBasic coordinate system (-80 to 79 in the x direction and -50 to 49 in the y direction). Suppose we wanted to move the sprite to the middle of the screen, we would do this…

```
3dMove 1,0,0,0
```

The ‘z’ parameter picks a bitmapped image out of the images within the sprite.

RMove command

The 3dRMove command positions your sprite (or camera) on the screen, but relative to the current position:

```
3dRMove n,r,theta,z
```

The variable ‘n’ is the sprite number (0 or “Camera” to specify the camera). ‘r’ is the distance to travel. ‘theta’ is the angle to move in (degrees) and z is the index of the bitmap to display. (In the case of moving the Camera, z is ignored).

Redraw

Nothing is shown on the screen until you execute the Redraw command. This command clears the display and redraws each wall and sprite in its new position.

```
3dRedraw
```

Collision Detection

The distance between sprites determines collisions. There is no command for detecting collisions between sprites and walls. To determine if two sprites are close together, use the 3dCollision function:

```
h=3dCollision(spr1, spr2, d)
```

spr1 is the first sprite (or 0 or “Camera” for the camera) and spr2 is the second sprite. ‘d’ is the distance between the sprites. If spr1 is within d pixels of spr2 then 3dCollision returns TRUE, else it returns FALSE.

Get Position

To retrieve the current position of a sprite (or the camera) use 3dGet

3dGet sprite_no, x, y, z

sprite_no is the number of the sprite to retrieve, x is the x coordinate, y is the y coordinate and z is the current bitmap displayed (or in the case of the Camera, z is the direction in degrees the camera is facing).

Example Program: 3d.app

Here is a sample program which will create a room with a twirling Cybiko logo. The arrow keys will allow you move around inside the room. If you bump into the walls you will be stopped. If you bump into the spinning logo the Cybiko will vibrate.

This complete application can be found in the "3d" directory:

```
OPTION C_COORDS
OPTION SHOW OFF
OPTION ESCAPE OFF

paper black

3droom 0;
3dwall -500, -500, 500, -500, "wall.tex"
3dwall  500, -500,  500,  500, "wall.tex"
3dwall  500,  500, -500,  500, "wall.tex"
3dwall -500,  500, -500, -500, "wall.tex"
3dmove 0,0,0,0
3dsprite 1,"column.spr"
3dmove 1,200,200,0
3dredraw
redraw show
dim x
dim y
dim speed
dim dir
dim hit
dim bgd
dim z

z=0
bgd=black
x=0
y=0
dir=0
speed=30

sub repaint
   z=(z+1) mod 19
   3dmove 1,200,200,z
   3dredraw
   printxy 0,0,"Speed: ", speed
   3dget camera,x,y,dir
   printxy 0,12,"Pos'n:", x, ",", y, ",", dir
   redraw show
end sub

while(true)
```

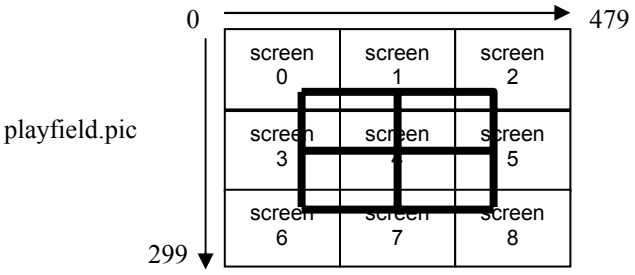
```
    inline _escape(0);
    if key(#KEY_RIGHT) then
       dir = dir - 10
       if dir < 0 then dir = 360+dir
       3dmove camera,x,y,dir
    endif
    if key(#KEY_LEFT) then
       dir = dir + 10
       if dir > 360 then dir = dir-360
       3dmove camera,x,y,dir
    endif
    if key(#KEY_UP) then
       3drmove camera,speed,dir,0
       3dget camera, x, y, dir
       if x>420 or x<-420 or y>420 or y<-420 then
             3drmove camera,-speed,dir,0
             3dget camera, x, y, dir
             beep 6
             wait 5
             beep -1
       endif
    endif
    if key(#KEY_DOWN) then
       3drmove camera,-speed,dir,0
       3dget camera,x, y, dir
       if x>420 or x<-420 or y>420 or y<-420 then
             3drmove camera,speed,dir,0
             3dget camera, x, y, dir
             beep 6
             wait 5
             beep -1
       endif
    endif
    if key(#KEY_DEL) then
       speed = speed - 10
    endif
    if key(#KEY_SPACE) then
       speed = speed + 10
    endif
    if key(#KEY_TAB) then
       bgd = bgd + 1
       if bgd = 4 then bgd = 0
       paper bgd
    end if
    repaint
    hit = 3dcollision(camera,1,10)
    vibrate 0
    if hit then vibrate 128
wend
```

Chapter 22: Playfield Graphics Part I : Introduction

The graphics behind such games as Zelda and Super Mario Brothers allows for a large playing field where the Cybiko screen is merely a window into the larger world. This type of graphics is complex and memory consuming. This tutorial will describe how to set up a 480x300 pixel playfield with an animated object and is the basis for just such games.

The Playfield:

A playfield is an extended bitmapped screen. In our case it will be 480x300 pixels. This is really 9 160x100 screens in a 3x3 array. The screens are called "screen0.bmp" through "screen8.bmp". They are combined into a single sprite in the "playfield.bld" file and are called "playfield.pic".



Our playfield is a highway system with roads leading from one screen to the next.

As the character (a car in our example) moves from one edge of the screen to the next, the next screen is displayed and the car is displayed in the appropriate place. This is handled automatically and with ease by the "repaint" routine listed below.

The Program

```
1. OPTION C_COORDS

2. dim x as int  'position of car from 0-480
3. dim y as int  'position of car from 0-300
4. dim z as int  'which image of car

5. dim sprite_field as int 'the sprite number of the playfield sprites
6. dim sprite_car as int   'the sprite number of the car sprites
7. dim car_left as int     'index into car sprite for going left
8. dim car_right as int    'index into car sprite for going right
9. dim car_up as int       'index into car sprite for going up
10.dim car_down as int     'index into car sprite for going down

11.'''
12.''' initialization
13.'''
14.sub init
15.sprite_field = 0  'playfield sprite
16.sprite_car = 1    'car sprite

17.car_left=0        'car left image
18.car_right=1       'car right image
```

```
19.car_up=2          'car up image
20.car_down=3        'car down image

21.'''
22.''' the playfield sprite has 9 bitmaps
23.''' one for each of the different areas of the
24.''' playfield
25.sprite sprite_field, "playfield.pic"

26.'''
27.''' the car has 4 bitmaps one for each direction
28.sprite sprite_car, "car.pic"

29.'''
30.''' the car can be anywhere on the playfield from
31.''' x = 0-480
32.''' y = 0-300
33.x=240  'center of the playfield
34.y=150  'center of the playfield
35.z=car_right
36.end sub

37.sub repaint
38.dim field_x as int   '0,1,2 for the horizontal bitmap
39.dim field_y as int   '0,1,2 for the vertical bitmap
40.dim field as int     '0-8 for the bitmap currently visible
41.dim car_x as int     '0-159 for the x coord on the screen
42.dim car_y as int     '0-99  for the y coord on the screen

43.field_x = x / 160    ' compute the horizontal bitmap
44.field_y = y / 100    ' compute the vertical bitmap
45.field = field_y*3 + field_x ' combine them to compute the bitmap
   number

46.car_x = x mod 160  ' compute the car's position on the screen
47.car_y = y mod 100  ' compute the car's position on the screen

48.move sprite_field, 0, 0, field  'display the current field
49.move sprite_car, car_x, car_y, z 'display the car

50.paper dkgrey  'transparent color for the car
51.redraw        'redraw all the sprites

52.end sub

53.'''
54.''' process the keyboard
55.'''
56.sub get_keys
57.if key(#KEY_UP) then   'move the car up
58.y=y-3
59.z=car_up
60.end if
61.if key(#KEY_DOWN) then 'move the car down
62.y=y+3
63.z=car_down
64.end if
65.if key(#KEY_LEFT) then 'move the car left
66.x=x-3
67.z=car_left
68.end if
```

```
69.if key(#KEY_RIGHT) then 'move the car right
70.x=x+3
71.z=car_right
72.end if
73.end sub

74.'''
75.''' main subroutine
76.'''
77.sub mainsub
78.init

79.while true
80.repaint
81.get_keys
82.wend
83.end sub

84.'''
85.''' call the main subroutine
86.'''
87.mainsub
```

Option C_Coords

The compiler option C_COORDS is used in this program to switch the coordinate system from CyBasic mode ([-80,-50] to [79,49]) into C mode ([0,0] to [159,99]). The zero-based coordinates make the mathematics of computing the screen coordinates easier. This is true for most nearly any video game.

The Variables

In our program we are extending the concept of the sprite position on the screen. The x and y coordinates define our position in the playfield – not our position on the Cybiko screen. So, the X variable can range from 0 (the leftmost position on the playfield) to 479 (the rightmost position on the playfield). Likewise the Y variable will range from 0 (top) to 299 (bottom). The Z variable is used to indicate which image of the car is currently being shown.

```
dim x as int  'position of car from 0-480
dim y as int  'position of car from 0-300
dim z as int  'which image of car
```

The remainder of the variables are actually constants used to make the program easier to read. SPRITE_FIELD and SPRITE_CAR are the sprite numbers for the playfield and the car respectively. The variables CAR_LEFT, CAR_RIGHT, CAR_UP, and CAR_DOWN are the indexes into the SPRITE_CAR sprite. These constants are initialized in the INIT function.

```
dim sprite_field as int 'the sprite number of the playfield
sprites
dim sprite_car as int   'the sprite number of the car sprites
dim car_left as int     'index into car sprite for going left
dim car_right as int    'index into car sprite for going right
dim car_up as int       'index into car sprite for going up
dim car_down as int     'index into car sprite for going down
```

sub init

Here we initialize all the “constants” in the program. Again, the SPRITE_FIELD and SPRITE_CAR are the sprite numbers. Later, when we actually load and move the sprites we will use these variables instead of

the literals “0” and “1”. This practice makes the code much easier to read. This is also true of the car’s left, right, up and down images. We use CAR_LEFT, etc. to make the program easier to read.

```
sprite_field = 0  'playfield sprite
sprite_car = 1    'car sprite

car_left=0        'car left image
car_right=1       'car right image
car_up=2          'car up image
car_down=3        'car down image
```

The playfield is a set of 9 bitmaps arranged in a 3x3 matrix. The car is a set of 4 bitmaps showing the car pointing left, right, up, and down. Here we load them into memory. Note that the playfield is 4K x 9 screens or 36K. Later we will investigate multiplayfield programs (where keeping all sprites in memory at one time will be impossible).

```
sprite sprite_field, "playfield.pic"
sprite sprite_car, "car.pic"
```

As I mentioned earlier, the playfield can be from (0,0) to (479,299). So, we position the car in the middle of the playfield with x=240 and y=150. And we start the car out pointing to the right. Remember all coordinates for x & y are in the playfield coordinate system. Later, in the REPAINT subroutine we will convert the playfield coordinates into Cybiko screen coordinates.

```
x=240  'center of the playfield
y=150  'center of the playfield
z=car_right
```

sub repaint

The repaint subroutine does most of the hard work in this program – and still the computations are fairly simple. The local variables FIELD_X and FIELD_Y are the x and y values of the 3x3 playfield matrix. So, the upper left corner of the playfield is 0,0 and the middle segment is 1,1. These values can then be combined to select one of the 9 bitmaps from the playfield sprite.

CAR_X and CAR_Y are the coordinates for the car on the Cybiko screen. Remember that X and Y are the position of the car on the playfield, so we need to convert the playfield coordinates into Cybiko coordinates.

```
dim field_x as int   '0,1,2 for the horizontal bitmap
dim field_y as int   '0,1,2 for the vertical bitmap
dim field as int     '0-8 for the bitmap currently visible
dim car_x as int     '0-159 for the x coord on the screen
dim car_y as int     '0-99  for the y coord on the screen
```

The calculation for finding the playfield screen is simple but uses some mathematical tricks. The first trick is that we are using the position of the car to determine which screen is to be displayed. The second trick is that the playfield screens are exactly the same size as the Cybiko screen. So, it is easy to determine which screen to display by dividing the car’s x coordinate by 160 (which will result in 0, 1, or 2) and the car’s y coordinate by 100 (which will also result in 0, 1, or 2). The final trick is that the playfield screens are stored in a special order from upper left corner to lower right corner (see picture above). So if we multiply the FIELD_Y variable by 3 and add the FIELD_X variable we get the correct playfield screen. Simple, huh?

```
field_x = x / 160    ' compute the horizontal bitmap
field_y = y / 100    ' compute the vertical bitmap
field = field_y*3 + field_x ' combine them to compute the bitmap
number
```

The car’s position on the screen is very simple to compute too. We just take the remainder of the X variable divided by 160. This is what the MOD function does. Likewise for the Y variable.

```
car_x = x mod 160  ' compute the car's position on the screen
car_y = y mod 100  ' compute the car's position on the screen
```

Since the car determines the screen to be displayed, moving the car around the screen makes a natural and easy interface for updating the playfield with the right screen.

Now all we have to do is display the sprites. The SPRITE_FIELD is displayed at 0,0 and the SPRITE_CAR is displayed at car_x and car_y.

```
move sprite_field, 0, 0, field  'display the current field
move sprite_car, car_x, car_y, z 'display the car
```

Finally we redraw the display. The "PAPER DKGREY" command sets the transparent color for the car. Otherwise we'd see a dark gray rectangle surrounding the car.

```
paper dkgrey  'transparent color for the car
redraw        'redraw all the sprites
```

sub get_keys

There is nothing special here. We check each key and move the car in one of 4 directions. There is plenty missing from this function. Like, what if we go off the left edge of the playfield? There is also nothing that keeps the car on the pavement. These capabilities will be covered in a future tutorial.

```
if key(#KEY_UP) then   'move the car up
      y=y-3
      z=car_up
end if
if key(#KEY_DOWN) then 'move the car down
      y=y+3
      z=car_down
end if
if key(#KEY_LEFT) then 'move the car left
      x=x-3
      z=car_left
end if
if key(#KEY_RIGHT) then 'move the car right
      x=x+3
      z=car_right
end if
```

sub mainsub

The main subrouting is "mainsub". It calls the initialization routine "init" and the proceeds to loop forever calling "repaint" and "get_keys".

Chapter 23: Playfield Graphics Part II: Animation

In Part-1 we learned how to create a 480x300-pixel playfield and move a car around. The car is not very interesting because it's not animated. In Part-2 I selected a lemming character to move around. Why? Because I happened to have an animated lemming in my library and no animated cars. The lemming is black and grey with a white background. This didn't fit the playfield I created in Part-1, so I inverted the playfield colors. So don't be shocked by the changes, it's still the same basic playfield logic as in Part-1. Now let's look at the animated lemming.

The Lemming:

The character sprite is composed of 4 sets of 5 bitmaps (20 in all). The 5 bitmaps comprise a walking lemming. Each bitmap is rotated to create a lemming walking in one of the four directions: Left, Right, Up, and Down. The images are stored in one sprite as "lemming.pic".

It's important to remember how the images are stored because we will be setting the 'z' parameter of the move command based on the direction the lemming is moving. Images 0-4 are the bitmaps of the lemming walking to the left. Images 5-9 are the bitmaps of the lemming walking to the right. Images 10-14 are the bitmaps of the lemming walking up. And finally images 15-19 are the bitmaps of the lemming walking down.

We'll keep track of the direction of the lemming with the Z_DIR variable. So, to show the lemming walking to the left, we set the Z_DIR variable to 0. To show the lemming walking to the right, Z_DIR is set to 5. For walking up, Z_DIR is set to 10. Finally, to show the lemming walking down, Z_DIR is set to 15.

To make the lemming walk, we add 0,1,2,3 or 4 to the Z_DIR variable. The Z_CNT variable will be constantly incremented from 0-4. When the lemming is moved, we set the move's 'z' parameter to Z_DIR+Z_CNT.

The Program

```
1. OPTION C_COORDS

2. dim x as int  'position of lemming from 0-480
3. dim y as int  'position of lemming from 0-300
4. dim z_dir as int  'which direction of lemming
5. dim z_cnt as int  'animation image of lemming

6. dim sprite_field as int 'the sprite number of the playfield sprites
7. dim sprite_lemming as int   'the sprite number of the lemming sprites
8. dim lemming_left as int     'index into lemming sprite for going left
9. dim lemming_right as int      'index into lemming sprite for going
   right
10.dim lemming_up as int       'index into lemming sprite for going up
11.dim lemming_down as int     'index into lemming sprite for going down

12.'''
13.''' initialization
14.'''
15.sub init
16.sprite_field = 0  'playfield sprite
17.sprite_lemming = 31     'lemming sprite is largest possible sprite -
   on top

18.lemming_left=0        'lemming left image
19.lemming_right=5       'lemming right image
20.lemming_up=10         'lemming up image
```

```
21.lemming_down=15       'lemming down image

22.'''
23.''' the playfield sprite has 9 bitmaps
24.''' one for each of the different areas of the
25.''' playfield
26.sprite sprite_field, "playfield.pic"

27.'''
28.''' the lemming has 20 bitmaps 5 for each direction
29.sprite sprite_lemming, "lemming.pic"

30.'''
31.''' the lemming can be anywhere on the playfield from
32.''' x = 0-480
33.''' y = 0-300
34.x=240  'center of the playfield
35.y=150  'center of the playfield
36.z_dir=lemming_right  ' lemming faces right
37.z_cnt=0              ' animation counter
38.end sub

39.sub repaint
40.dim field_x as int   '0,1,2 for the horizontal bitmap
41.dim field_y as int   '0,1,2 for the vertical bitmap
42.dim field as int     '0-8 for the bitmap currently visible
43.dim lemming_x as int     '0-159 for the x coord on the screen
44.dim lemming_y as int     '0-99  for the y coord on the screen

45.field_x = x / 160    ' compute the horizontal bitmap
46.field_y = y / 100    ' compute the vertical bitmap
47.field = field_y*3 + field_x ' combine them to compute the bitmap
   number

48.lemming_x = x mod 160  ' compute the lemming's position on the screen
49.lemming_y = y mod 100  ' compute the lemming's position on the screen

50.move sprite_field, 0, 0, field  'display the current field
51.move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
   lemming

52.paper white  'transparent color for the lemming
53.redraw        'redraw all the sprites

54.end sub

55.'''
56.''' increment the z_cnt variable
57.'''
58.sub update_z_cnt
59.z_cnt=(z_cnt+1) MOD 5
60.end sub

61.'''
62.''' process the keyboard
63.'''
64.sub get_keys
65.if key(#KEY_UP) then   'move the lemming up
66.y=y-3
67.z_dir=lemming_up
68.update_z_cnt
```

```
69.end if
70.if key(#KEY_DOWN) then 'move the lemming down
71.y=y+3
72.z_dir=lemming_down
73.update_z_cnt
74.end if
75.if key(#KEY_LEFT) then 'move the lemming left
76.x=x-3
77.z_dir=lemming_left
78.update_z_cnt
79.end if
80.if key(#KEY_RIGHT) then 'move the lemming right
81.x=x+3
82.z_dir=lemming_right
83.update_z_cnt
84.end if
85.end sub

86.'''
87.''' main subroutine
88.'''
89.sub mainsub
90.init

91.while true
92.repaint
93.get_keys
94.wend
95.end sub

96.'''
97.''' call the main subroutine
98.'''
99.mainsub
```

Option C_Coords

As with last time, the compiler option C_COORDS is used to switch the coordinate system from CyBasic mode ([-80,-50] to [79,49]) into C mode ([0,0] to [159,99]). The zero-based coordinates make the mathematics of computing the screen coordinates easier. This is true for most nearly any video game.

The Variables

As in Part-1 the x and y variables designate the lemming's position on the playfield. However, this time, instead of a single 'z' variable, we have two new variables. Z_DIR will hold a value of 0, 5, 10, or 15 – for the direction the lemming is facing. Z_CNT will range from 0-4 for the image of the lemming walking.

```
dim x as int  'position of car from 0-480
dim y as int  'position of car from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming
```

The remainder of the variables are actually constants used to make the program easier to read. SPRITE_FIELD and SPRITE_LEMMING are the sprite numbers for the playfield and the lemming respectively. The variables LEMMING_LEFT, LEMMING_RIGHT, LEMMING_UP, and LEMMING_DOWN are the indexes into the SPRITE_LEMMING sprite. These constants are initialized in the INIT function.

```
dim sprite_field as int
```

```
dim sprite_lemming as int
dim lemming_left as int
dim lemming_right as int
dim lemming_up as int
dim lemming_down as int
```

sub init

Here we initialize all the "constants" in the program. Again, the SPRITE_FIELD and SPRITE_LEMMING are the sprite numbers. Later, when we actually load and move the sprites we will use these variables instead of the literals "0" and "31".

Notice that we have set the lemming up as sprite 31. This puts the lemming "on top" of all other sprites. As you may recall, sprite 0 is the background sprite and all other sprites are painted on top of it. The sprites are painted starting with 0 and on up to 31.
So, since sprite 31 is painted last, it appears on top of all others.

```
sprite_field = 0  'playfield sprite
sprite_lemming = 31 'lemming is largest possible sprite - on top
```

```
LEMMING_LEFT is the index into the lemming sprite.  LEMMING_LEFT is the
first sprite that looks to the left. Likewise, LEMMING_RIGHT points to
the first sprite that looks right.   And so on with LEMMING_UP and
LEMMING_DOWN.
```

```
lemming_left=0        'lemming left image
lemming_right=5       'lemming right image
lemming_up=10         'lemming up image
lemming_down=15       'lemming down image
```

Here we load the sprites into memory.

```
sprite sprite_field, "playfield.pic"
sprite sprite_lemming, "lemming.pic"
```

As in Part-1, we position the lemming in the middle of the playfield by positioning it at (240,150). The Z_DIR variable is set to the LEMMING_RIGHT value, and Z_CNT is set to zero. Later, Z_DIR and Z_CNT will be added together to determine the bitmap to display on the screen.

```
x=240  'center of the playfield
y=150  'center of the playfield
z_dir=lemming_right  ' lemming faces right
z_cnt=0              ' animation counter
```

sub repaint

As in Part-1, the repaint routine selects the playfield screen to display and positions the character on the screen. First we define the variables

```
dim field_x as int   '0,1,2 for the horizontal bitmap
dim field_y as int   '0,1,2 for the vertical bitmap
dim field as int     '0-8 for the bitmap currently visible
dim lemming_x as int     '0-159 for the x coord on the screen
dim lemming_y as int     '0-99  for the y coord on the screen
```

Here we compute the playfield screen.

```
field_x = x / 160    ' compute the horizontal bitmap
field_y = y / 100    ' compute the vertical bitmap
```

```
field = field_y*3 + field_x ' combine them to compute the bitmap
number
```

Here we compute the position of the lemming on the screen.

```
lemming_x = x mod 160   ' compute the lemming's position on the
screen
lemming_y = y mod 100   ' compute the lemming's position on the
screen
```

Now we position the playfield sprite on the screen.

```
move sprite_field, 0, 0, field  'display the current field
```

This is the "magic" – the 'z' parameter is computed as the Z_DIR variable plus the Z_CNT variable. Z_DIR is the pointer to the first image of Left, Right, Up, Down. Z_CNT is a value from 0-4 that selects which set of "feet" the lemming will display for the current direction we are going.

```
move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display
the lemming
```

Finally we set the transparent color and redraw all the sprites on the screen.

```
paper white  'transparent color for the lemming
redraw        'redraw all the sprites
```

sub update_z_cnt

This very short subroutine is designed to march Z_CNT from 0 to 4 and back again. Since Z_CNT is the animation counter it mustn't go over 4. Here we use a little mathematical magic. MOD returns the remainder of a division. So dividing by 5 will always have a remainder from 0-4.

```
z_cnt=(z_cnt+1) MOD 5
```

sub get_keys

This function operates similarly to the one in Part-1. When a key is pressed, the Z_DIR variable is pointed to the first lemming pointing in the direction of the key. Then, the UPDATE_Z_CNT subroutine is called to increment the animation variable, Z_CNT. And of course the appropriate X or Y variable is incremented or decremented to move the lemming around. Still we are not restricting the lemming to the road, that will happen in the next tutorial.

```
if key(#KEY_UP) then   'move the lemming up
      y=y-3
      z_dir=lemming_up
      update_z_cnt
end if
if key(#KEY_DOWN) then 'move the lemming down
      y=y+3
      z_dir=lemming_down
      update_z_cnt
end if
if key(#KEY_LEFT) then 'move the lemming left
      x=x-3
      z_dir=lemming_left
      update_z_cnt
end if
if key(#KEY_RIGHT) then 'move the lemming right
      x=x+3
      z_dir=lemming_right
      update_z_cnt
end if
```

sub mainsub

The main subroutine is “mainsub”. It calls the initialization routine “init” and the proceeds to loop forever calling “repaint” and “get_keys”.

Chapter 24: Playfield Graphics Part III: Background Collisions

In Part-1 we learned how to create a 480x300-pixel playfield and move a car around. In part two, we replaced the car with an animated lemming. While the program is more interesting, it lacks a certain "reality" – we can still walk all over the screen. Part-3 of the Playfield Graphics Tutorial introduces the concept of Background Collision Detection. This is the logic that will constrain our lemming to walk only on the road.

The Collision Table:

As you will recall, the playfield is 480x300 pixels. One way to constrain the lemming to walk only on the road is to look at the color of the pixel the lemming is standing on. If the pixel is white then the lemming can walk there. If the pixel is any other color, then the lemming cannot. This works fine except that the lemming would be bumping into "safe" objects like the dotted line in the road.

A better way would be to have a separate bitmap of all the regions the lemming is allowed to walk on. This bitmap would have a black pixel for regions the lemming is not allowed to walk on, and a white pixel for every region it is allowed to walk on. This would take a bitmap of 480x300 pixels – or 36K of memory. Not very economical.

When you look at the playfield, its clear that large sections are "legal" to walk on. You might be able to represent a rectangular region by a single yes or no value. Thus create a table of legal/illegal regions. This is just what we do with the Collision Table.

The Collision Table is a two-dimensional array of characters. It is 30x30 characters – just 900 bytes. Each element of the array represents a 16x10-pixel region on the playfield. If the array element has an 'x' in it, then the lemming is allowed to walk on that part of the playfield. If the array element has a '.' in it, the lemming is not allowed to walk there.

So, if COLLISION_TABLE[0,0] has an 'x' in it, then the lemming is allowed to walk on pixels in the rectangle represented by pixels (0,0) through (15,9).

The Collision Table is initialized with a series of assignments. Each one looks like this:

collision_table[4]="..xxxxxxxxxxxxxxxxxxxxxxxxx..." 'y=40-49

Here, each '.' represents a region the lemming is not allowed to walk, and each 'x' is a region where it is. There is a new subroutine called "INIT_PLAYFIELD()" which sets up this table.

Some changes have to be made to the GET_KEYS() logic so that whenever we hit a key, we check to make sure the lemming is allowed to move in that direction. If moving in that direction puts the lemming in an illegal region, the move is ignored. And of course, if moving in that direction puts the lemming in a legal region, then the move is performed as usual.

The Program

```
'''
''' playfield tutorial - part 3
''' Simple Collision Detection
'''
''' by greg smith
'''

'''
''' use c coordinates
'''
OPTION C_COORDS

dim x as int  'position of lemming from 0-480
dim y as int  'position of lemming from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming

dim sprite_field as int 'the sprite number of the playfield sprites
dim sprite_lemming as int   'the sprite number of the lemming sprites
dim lemming_left as int     'index into lemming sprite for going left
dim lemming_right as int    'index into lemming sprite for going right
dim lemming_up as int       'index into lemming sprite for going up
dim lemming_down as int     'index into lemming sprite for going down

dim collision_street as char     'the street's collision value

dim collision_table[30,31] as char 'the collision table

'''
''' initialization
'''
sub init
    sprite_field = 0  'playfield sprite
    sprite_lemming = 31    'lemming sprite is largest possible sprite -
on top

    lemming_left=0        'lemming left image
    lemming_right=5       'lemming right image
    lemming_up=10         'lemming up image
    lemming_down=15       'lemming down image

    '''
    ''' the playfield sprite has 9 bitmaps
    ''' one for each of the different areas of the
    ''' playfield
    sprite sprite_field, "playfield.pic"

    '''
    ''' the lemming has 20 bitmaps 5 for each direction
    sprite sprite_lemming, "lemming.pic"

    '''
    ''' the lemming can be anywhere on the playfield from
    ''' x = 0-480
    ''' y = 0-300
    x=240  'center of the playfield
    y=150  'center of the playfield
    z_dir=lemming_right  ' lemming faces right
```

```
    z_cnt=0             ' animation counter

    '''
    ''' set the collision flags
    '''
    collision_street = \x\
end sub

sub repaint
    dim field_x as int   '0,1,2 for the horizontal bitmap
    dim field_y as int   '0,1,2 for the vertical bitmap
    dim field as int     '0-8 for the bitmap currently visible
    dim lemming_x as int     '0-159 for the x coord on the screen
    dim lemming_y as int     '0-99  for the y coord on the screen

    field_x = x / 160    ' compute the horizontal bitmap
    field_y = y / 100    ' compute the vertical bitmap
    field = field_y*3 + field_x ' combine them to compute the bitmap
number

    lemming_x = x mod 160   ' compute the lemming's position on the
screen
    lemming_y = y mod 100   ' compute the lemming's position on the
screen

    move sprite_field, 0, 0, field  'display the current field
    move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
lemming

    paper white  'transparent color for the lemming
    ink ltgrey

    redraw        'redraw all the sprites

end sub

'''
''' increment the z_cnt variable
'''
sub update_z_cnt
    z_cnt=(z_cnt+1) MOD 5
end sub

'''
''' check for collision against table
'''
function collision_check(x as int, y as int) as char
    collision_check = collision_table[y/10,x/16]
end function

'''
''' process the keyboard
'''
sub get_keys
    dim tmp as int
    if key(#KEY_UP) then   'move the lemming up
        tmp=y-3
        if collision_check(x,tmp) = collision_street then
            y=tmp
            z_dir=lemming_up
            update_z_cnt
```

```
        end if
    end if
    if key(#KEY_DOWN) then 'move the lemming down
        tmp=y+3
        if collision_check(x,tmp) = collision_street then
            y=tmp
            z_dir=lemming_down
            update_z_cnt
        end if
    end if
    if key(#KEY_LEFT) then 'move the lemming left
        tmp=x-3
        if collision_check(tmp,y) = collision_street then
            x=tmp
            z_dir=lemming_left
            update_z_cnt
        end if
    end if
    if key(#KEY_RIGHT) then 'move the lemming right
        tmp=x+3
        if collision_check(tmp,y) = collision_street then
            x=tmp
            z_dir=lemming_right
            update_z_cnt
        end if
    end if
end sub

sub init_playfield(field as int)
    collision_table[ 0]="............................." 'y=0-9
    collision_table[ 1]="............................." 'y=10-19
    collision_table[ 2]="............................." 'y=20-29
    collision_table[ 3]="............................." 'y=30-39
    collision_table[ 4]="..xxxxxxxxxxxxxxxxxxxxxxxxx..." 'y=40-49
    collision_table[ 5]="..x..........x..........x..." 'y=50-59
    collision_table[ 6]="..x..........x..........x..." 'y=60-69
    collision_table[ 7]="..x..........x..........x..." 'y=70-79
    collision_table[ 8]="..x..........x..........x..." 'y=80-89
    collision_table[ 9]="..x..........x..........x..." 'y=90-99
    collision_table[10]="..x..........x..........x..." 'y=100-109
    collision_table[11]="..x..........x..........x..." 'y=110-119
    collision_table[12]="..x..........x..........x..." 'y=120-129
    collision_table[13]="..x..........x..........x..." 'y=130-139
    collision_table[14]="..xxxxxxxxxxxxxxxxxxxxxxxxx..." 'y=140-149
    collision_table[15]="..x..........x..........x..." 'y=150-159
    collision_table[16]="..x..........x..........x..." 'y=160-169
    collision_table[17]="..x..........x..........x..." 'y=170-179
    collision_table[18]="..x..........x..........x..." 'y=180-189
    collision_table[19]="..x..........x..........x..." 'y=190-199
    collision_table[20]="..x..........x..........x..." 'y=200-209
    collision_table[21]="..x..........x..........x..." 'y=210-219
    collision_table[22]="..x..........x..........x..." 'y=220-229
    collision_table[23]="..x..........x..........x..." 'y=230-239
    collision_table[24]="..xxxxxxxxxxxxxxxxxxxxxxxxx..." 'y=240-249
    collision_table[25]="............................." 'y=250-259
    collision_table[26]="............................." 'y=260-269
    collision_table[27]="............................." 'y=270-279
    collision_table[28]="............................." 'y=280-289
    collision_table[29]="............................." 'y=290-299
end sub
```

```
'''
''' main subroutine
'''
sub mainsub
    init
    init_playfield(0)

    while true
        repaint
        get_keys
    wend
end sub

'''
''' call the main subroutine
'''
mainsub
```

The Variables

Two new variables are introduced. The COLLISION_STREET variable is a constant. It is initialized to 'x' – the character representing a valid region for the lemming to walk upon. COLLISION_TABLE is a 30x31-character array. Note that while we need only 30x30 characters we allocate an extra byte per row – that is because strings require a null-terminator byte at the end of each string.

```
dim collision_street as char     'the street's collision value
dim collision_table[30,31] as char 'the collision table
```

sub init

The only change here is the initialization of the COLLISION_STREET flag.

```
collision_street = \x\
```

sub repaint

Amazingly, there were no changes to this function.

sub update_z_cnt

No changes here.

function collision_check

This function checks the collision table for a value and returns it. Notice the division of the Y variable by 10 and the X variable by 16. This divides the entire 480x300-pixel playfield into a 30x30 region. Also notice that the X and Y values are reversed from the normal convention. This is because the Y value is the row selector of the array and the X value is the column selector.

```
function collision_check(x as int, y as int) as char
    collision_check = collision_table[y/10,x/16]
end function
```

sub get_keys

Sub get_keys() performs pretty much as it did before, with an important difference. We first compute the location of the lemming and store it in a TMP variable. It is important to store it in TMP because we don't want to modify the position until we have verified it is a legal move.

Here, in the case of the UP key, we compute TMP=Y-3. Then we call COLLISION_CHECK() to determine if the square **we would move to** is on the street (COLLISION_STREET). If it is, then we update the Y variable, and do all the things we would normally do when moving (update Z_DIR and Z_CNT).

This same logic applies to moving in the other directions as well.

```
      dim tmp as int
          if key(#KEY_UP) then   'move the lemming up
              tmp=y-3
              if collision_check(x,tmp) = collision_street then
                  y=tmp
                  z_dir=lemming_up
                  update_z_cnt
              end if
          end if
```

sub init_playfield

INIT_PLAYFIELD is actually going to grow in the near future. Its job right now is to initialize the COLLISION_TABLE. But soon it will be a portal to adding multiple playfields.

The purpose of the COLLISION_TABLE is to have 1 character represent a 16x10-pixel region on the screen. Each character has either a '.' or an 'x' to represent where the lemming is allowed to roam. If you look at the code below, you should see a small resemblance to the actual playfield. This is because it is a reduced image of the playfield.

To create this table, I first inspected the bitmaps in Paint Shop Pro V5. With it I was able to determine the actual pixel locations of the road in the playfield. I then divided the x coordinates by 16. That told me what column the 'x' would belong in - in the table below. Likewise, I divided the y coordinates by 10 and it told me which row it would belong in.

Frankly, there is a bit of error in this process. If you plan your playfield out in advance you can arrange it so all walkways are on 16x10 boundaries. In the case of our playfield, you can see that the top and bottom roads allow the lemming to walk just over the line. This can easily be fixed by editing the bitmaps in a paint program.

```
collision_table[ 0]="............................." 'y=0-9
collision_table[ 1]="............................." 'y=10-19
collision_table[ 2]="............................." 'y=20-29
collision_table[ 3]="............................." 'y=30-39
collision_table[ 4]="..xxxxxxxxxxxxxxxxxxxxxxxx..." 'y=40-49
collision_table[ 5]="..x..........x..........x..." 'y=50-59
collision_table[ 6]="..x..........x..........x..." 'y=60-69
collision_table[ 7]="..x..........x..........x..." 'y=70-79
collision_table[ 8]="..x..........x..........x..." 'y=80-89
collision_table[ 9]="..x..........x..........x..." 'y=90-99
collision_table[10]="..x..........x..........x..." 'y=100-109
collision_table[11]="..x..........x..........x..." 'y=110-119
collision_table[12]="..x..........x..........x..." 'y=120-129
collision_table[13]="..x..........x..........x..." 'y=130-139
collision_table[14]="..xxxxxxxxxxxxxxxxxxxxxxxx..." 'y=140-149
collision_table[15]="..x..........x..........x..." 'y=150-159
collision_table[16]="..x..........x..........x..." 'y=160-169
collision_table[17]="..x..........x..........x..." 'y=170-179
collision_table[18]="..x..........x..........x..." 'y=180-189
collision_table[19]="..x..........x..........x..." 'y=190-199
collision_table[20]="..x..........x..........x..." 'y=200-209
collision_table[21]="..x..........x..........x..." 'y=210-219
```

```
collision_table[22]="..x..........x..........x..." 'y=220-229
collision_table[23]="..x..........x..........x..." 'y=230-239
collision_table[24]="..xxxxxxxxxxxxxxxxxxxxxxxx..." 'y=240-249
collision_table[25]="............................" 'y=250-259
collision_table[26]="............................" 'y=260-269
collision_table[27]="............................" 'y=270-279
collision_table[28]="............................" 'y=280-289
collision_table[29]="............................" 'y=290-299
```

sub mainsub

The mainsub now calls the INIT_PLAYFIELD subroutine.

Chapter 25: Playfield Graphics Part IV: Playfields as Data Files

In Part-1 we learned how to create a 480x300-pixel playfield and move a car around. In part two, we replaced the car with an animated lemming. Part-3 of the Playfield Graphics Tutorial introduced the concept of Background Collision Detection. Part-4 demonstrates how to put the collision data into a file.

This step by itself does not appear that significant, but it is a stepping stone to bigger things. In particular, this is a step towards having multiple playfields in a single program. A later tutorial will demonstrate how to use this feature to include multiple playfields.

The Collision Table:

Remember that the Collision Table is a two-dimensional array of characters. It is 30x30 characters – just 900 bytes. Each element of the array represents a 16x10-pixel region on the playfield. If the array element has an 'x' in it, then the lemming is allowed to walk on that part of the playfield. If the array element has a '.' in it, the lemming is not allowed to walk there.

In Part-3 we used explicit assignment statements to set up the table. In Part-4 we have put the name of the playfield sprite into a data file along with the collision table information. And, we have written code to read the collision table information into memory.

The Playfield.dat file:

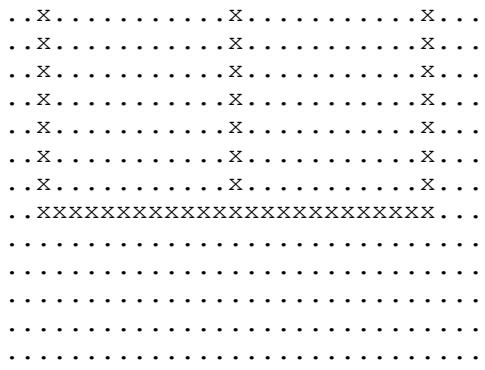
This new file is the "playfield.dat" file and is an ordinary text file you create and edit on the PC. The first line of the file is the name of the playfield sprite. Each of the remaining 30 lines is the Collision Table, just as we defined it in Part-3.

Adding the file to the application

This file is placed into the application by adding the following 2 lines into the ***playfield.bld*** file:
[files]
playfield1.dat

Playfield1.dat:

```
playfield.pic
..............................
..............................
..............................
..............................
..xxxxxxxxxxxxxxxxxxxxxxxxxx...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..x...........x...........x...
..xxxxxxxxxxxxxxxxxxxxxxxxxx...
..x...........x...........x...
..x...........x...........x...
```


..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..xxxxxxxxxxxxxxxxxxxxxxxxx...
....
....
....
....
....

The Program

```
'''
''' playfield tutorial - part 4
''' putting the playfield in a file
'''
''' by greg smith
'''

'''
''' use c coordinates
'''
OPTION C_COORDS

'''
''' turn off escape processing for speed
'''
OPTION ESCAPE OFF

dim x as int  'position of lemming from 0-480
dim y as int  'position of lemming from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming

dim sprite_field as int 'the sprite number of the playfield sprites
dim sprite_lemming as int   'the sprite number of the lemming sprites
dim lemming_left as int     'index into lemming sprite for going left
dim lemming_right as int    'index into lemming sprite for going right
dim lemming_up as int       'index into lemming sprite for going up
dim lemming_down as int     'index into lemming sprite for going down

dim collision_street as char     'the street's collision value

dim collision_table[30,31] as char 'the collision table

'''
''' update file - copy file from app to flash
'''
sub update_file(fname[] as char)
    inline File_remove(fname);
    inline _load(fname);
end sub

'''
''' initialization
'''
sub init
    update_file("playfield1.dat") 'get the .dat file into flash


    sprite_field = 0  'playfield sprite
    sprite_lemming = 31    'lemming sprite is largest possible sprite -
on top

    lemming_left=0        'lemming left image
    lemming_right=5       'lemming right image
    lemming_up=10         'lemming up image
    lemming_down=15       'lemming down image

    '''
```

```
    ''' the lemming has 20 bitmaps 5 for each direction
    sprite sprite_lemming, "lemming.pic"

    '''
    ''' the lemming can be anywhere on the playfield from
    ''' x = 0-480
    ''' y = 0-300
    x=240  'center of the playfield
    y=150  'center of the playfield
    z_dir=lemming_right  ' lemming faces right
    z_cnt=0              ' animation counter

    '''
    ''' set the collision flags
    '''
    collision_street = \x\
end sub

sub repaint
    dim field_x as int   '0,1,2 for the horizontal bitmap
    dim field_y as int   '0,1,2 for the vertical bitmap
    dim field as int     '0-8 for the bitmap currently visible
    dim lemming_x as int     '0-159 for the x coord on the screen
    dim lemming_y as int     '0-99  for the y coord on the screen

    field_x = x / 160    ' compute the horizontal bitmap
    field_y = y / 100    ' compute the vertical bitmap
    field = field_y*3 + field_x ' combine them to compute the bitmap
number

    lemming_x = x mod 160   ' compute the lemming's position on the
screen
    lemming_y = y mod 100   ' compute the lemming's position on the
screen

    move sprite_field, 0, 0, field  'display the current field
    move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
lemming

    paper white  'transparent color for the lemming
    ink ltgrey

    redraw        'redraw all the sprites

end sub

'''
''' increment the z_cnt variable
'''
sub update_z_cnt
    z_cnt=(z_cnt+1) MOD 5
end sub

'''
''' check for collision against table
'''
function collision_check(x as int, y as int) as char
    collision_check = collision_table[y/10,x/16]
end function

'''
```

```
''' process the keyboard
'''
sub get_keys
    dim tmp as int
    if key(#KEY_UP) then   'move the lemming up
        tmp=y-3
        if collision_check(x,tmp) = collision_street then
            y=tmp
            z_dir=lemming_up
            update_z_cnt
        end if
    end if
    if key(#KEY_DOWN) then 'move the lemming down
        tmp=y+3
        if collision_check(x,tmp) = collision_street then
            y=tmp
            z_dir=lemming_down
            update_z_cnt
        end if
    end if
    if key(#KEY_LEFT) then 'move the lemming left
        tmp=x-3
        if collision_check(tmp,y) = collision_street then
            x=tmp
            z_dir=lemming_left
            update_z_cnt
        end if
    end if
    if key(#KEY_RIGHT) then 'move the lemming right
        tmp=x+3
        if collision_check(tmp,y) = collision_street then
            x=tmp
            z_dir=lemming_right
            update_z_cnt
        end if
    end if
end sub

'''
''' get a string from the file
''' a string ends in CR/LF
'''
sub getstring(f as int, s[] as char)
    dim x as char
    dim n as int

    n=0

    while(true)
        get f,,x
        if x<>13 then
            if x=10 then exit while
            s[n] = x
            s[n+1] = 0
            n=n+1
        endif
    wend
end sub

'''
''' read the playfield file defined by
```

```
''' 'field' and set up the sprites and
''' the collision table
'''
sub init_playfield(field as int)
    dim fname[32] as char
    dim s[32] as char

    cls
    print "Loading..."

    ''' create the filename to read
    sprint fname, "playfield", field, ".dat"

    ''' open the playfield definition file
    open fname for read as 1

    getstring(1, s) 'get the sprite name

    '''
    ''' the playfield sprite has 9 bitmaps
    ''' one for each of the different areas of the
    ''' playfield
    sprite sprite_field, s

    '''
    ''' read the 30 playfield collision rows
    '''
    for i=0 to 29
        getstring(1, collision_table[i])
    next

    '''
    ''' close the playfield.dat file
    '''
    close 1
end sub

'''
''' main subroutine
'''
sub mainsub
    init
    init_playfield(1)

    while true
        inline _escape(0); /* check the escape key */
        repaint
        get_keys
    wend
end sub

'''
''' call the main subroutine
'''
mainsub
```

The Variables

No new variables are introduced.

Option Escape Off

Normally, B2C inserts a check for the ESC key at the top of every loop. This check takes 1/10 of a second to perform. For simple programs, this is not a problem. But as a program becomes more advanced, there are more and more loops and the ESC key check becomes a serious performance bottleneck. Specifying OPTION ESCAPE OFF tells B2C not to do ESC checking. Thus enhancing performance. However, it is now up to the programmer to insert the following line of code into their program at the top of a major loop:

inline _escape(0);

This will check the ESC key and display a dialog box if the key is detected.

sub update_file

Update_file() will remove the specified file from the flash memory of the Cybiko and replace it with the same file from inside the application's archive. This is useful for files like .dl's and data files (which must be in the flash memory to be opened by B2C).

The first command "inline File_remove(fname)" is a call to the Cybiko OS to remove the file in question from the flash. The second command "inline _load(fname)" is a call to the B2C runtime library to copy a file from the archive into flash memory.

We will use this command to copy the "playfield1.dat" file from the application archive into flash memory.

sub init

The sprite command defining the playfield has been moved to the ***init_playfield()*** function.

sub repaint

there were no changes to this function.

sub update_z_cnt

No changes here.

function collision_check

No changes here.

sub get_keys

No changes here.

sub getstring

This subroutine reads from an opened file until a carriage-return / line-feed is detected and returns the string in a variable. This is a very handy function which you may want to reuse in other programs.

sub init_playfield

This function has changed significantly. We construct the filename by appending the *field* variable to the word "playfield" and then add on a ".dat". Thus, when *field*=1 the filename becomes "playfield1.dat".

Then, we open a file with that name for read. We read the first line of the file to get the sprite name of the playfield. Finally we call the *sprite* command to load the sprite into memory.

Next we read the 30 lines of collision table data into the collision table.

And lastly we close the file and return.

sub mainsub

The mainsub now calls the INIT_PLAYFIELD subroutine with a parameter of '1'. Additionally, it calls the "_escape(0)" function from the C library using the *inline* command. This tests the keyboard for the ESC key. We do this just once at the top of the loop because it takes about 1/10 of a second to do this check. Checking at the top of each loop (see OPTION ESCAPE description, above) takes too much time and slows the program perceptibly.

Chapter 26: Playfield Graphics Part V: Multiple Playfields

Part-3 of the Playfield Graphics Tutorial introduced the concept of Background Collision Detection. Part-4 demonstrates how to put the collision data into a file. In Part-5 we implement multiple playfields to extend the game play.

The New Playfield:

The new playfield (Playfield-2) is another set of roads, but this time the roads have a different set of turns than the first set – and they are in a city instead of a highway. The bitmaps in the second set of roads are named street0.bmp – street8.bmp. Remember to add them to the playfield.bld file:

```
[pic=street.pic]
street0.bmp
street1.bmp
street2.bmp
street3.bmp
street4.bmp
street5.bmp
street6.bmp
street7.bmp
street8.bmp
```

The Playfield.dat file:

In Part-4 we created the playfield.dat file with the name of the playfield and the collision detection data. In Part-5 we create a second playfield, and a playfield.dat file for it called playfield2.dat.

In addition, we add a new character to the playfield.dat file. In Part-3 we used the period ('.') to indicate an area the lemming could not walk, and the 'x' to indicate areas where the lemming could walk. Now, we also use the digits 1-9 to represent alternate playfields. If a collision table entry has a '2' in it, it means the lemming will be transported to playfield-2. Or, if a collision table entry has a '1' in it, it means the lemming will be transported back to playfield –1.

To facilitate this, a castle icon has been added to the playfields to indicate a portal from one playfield to the other. In playfield-1 this portal is on screen5.bmp. In playfield-2 its on street0.bmp. Whenever the lemming walks into the castle it will be transported to the center of the other playfield.

The playfield 2.dat file must be added to the playfield.bld file:

```
[files]
playfield1.dat
playfield2.dat
```

Playfield1.dat:

Notice the '2' character in the center-right of the collision map. This indicates the portal to transport from playfield-1 to playfield-2. When the lemming encounters this region of the screen, it will automatically be transported to the center of playfield-2.

```
playfield.pic
.............................
.............................
.............................
.............................
```

```
..xxxxxxxxxxxxxxxxxxxxxxxx...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..xxxxxxxxxxxxxxxxxxxxxxxx2...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..x..........x..........x...
..xxxxxxxxxxxxxxxxxxxxxxxx...
.............................
.............................
.............................
.............................
.............................
```

Playfield2.dat:

Notice the character ‘1’ at the top left of the playfield collision table. This indicates a portal from playfield-2 to playfield-1.

```
street.pic
.............................
.............................
.1xxxxxxxxxxxxxxxxxxxxxxx.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....x.........x.........x.....
....xxxxxxxxxxxxxxxxxxxxxx.....
.............................
.............................
.............................
.............................
```

The Program

```
'''
''' playfield tutorial - part 5
''' multiple playfields
'''
''' by greg smith
'''

'''
''' use c coordinates
'''
OPTION C_COORDS

'''
''' turn off escape processing for speed
'''
OPTION ESCAPE OFF

dim x as int  'position of lemming from 0-480
dim y as int  'position of lemming from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming

dim sprite_field as int 'the sprite number of the playfield sprites
dim sprite_lemming as int   'the sprite number of the lemming sprites
dim lemming_left as int     'index into lemming sprite for going left
dim lemming_right as int    'index into lemming sprite for going right
dim lemming_up as int       'index into lemming sprite for going up
dim lemming_down as int     'index into lemming sprite for going down

dim collision_street as char     'the street's collision value
dim collision_pf0 as char        'playfield 0
dim collision_pf9 as char        'playfield 9

dim collision_table[30,31] as char 'the collision table

'''
''' update file - copy file from app to flash
'''
sub update_file(fname[] as char)
    inline File_remove(fname);
    inline _load(fname);
end sub

'''
''' initialization
'''
sub init
    update_file("playfield1.dat") 'get the .dat file into flash
    update_file("playfield2.dat")

    sprite_field = 0  'playfield sprite
    sprite_lemming = 31    'lemming sprite is largest possible sprite -
on top

    lemming_left=0        'lemming left image
    lemming_right=5       'lemming right image
    lemming_up=10         'lemming up image
    lemming_down=15       'lemming down image
```

```
    '''
    ''' the lemming has 20 bitmaps 5 for each direction
    sprite sprite_lemming, "lemming.pic"

    '''
    ''' the lemming can be anywhere on the playfield from
    ''' x = 0-480
    ''' y = 0-300
    x=240  'center of the playfield
    y=150  'center of the playfield
    z_dir=lemming_right  ' lemming faces right
    z_cnt=0              ' animation counter

    '''
    ''' set the collision flags
    '''
    collision_street = \x\
    collision_pf0 = \0\
    collision_pf9 = \9\
end sub

sub repaint
    dim field_x as int   '0,1,2 for the horizontal bitmap
    dim field_y as int   '0,1,2 for the vertical bitmap
    dim field as int     '0-8 for the bitmap currently visible
    dim lemming_x as int     '0-159 for the x coord on the screen
    dim lemming_y as int     '0-99  for the y coord on the screen

    field_x = x / 160    ' compute the horizontal bitmap
    field_y = y / 100    ' compute the vertical bitmap
    field = field_y*3 + field_x ' combine them to compute the bitmap
number

    lemming_x = x mod 160   ' compute the lemming's position on the
screen
    lemming_y = y mod 100   ' compute the lemming's position on the
screen

    move sprite_field, 0, 0, field  'display the current field
    move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
lemming

    paper white  'transparent color for the lemming
    ink ltgrey

    redraw        'redraw all the sprites

end sub

'''
''' increment the z_cnt variable
'''
sub update_z_cnt
    z_cnt=(z_cnt+1) MOD 5
end sub

'''
''' check for collision against table
'''
function collision_check(x as int, y as int) as char
```

```
    collision_check = collision_table[y/10,x/16]
end function

'''
''' transport lemming to another playfield
'''
sub transport(f as int)
    init_playfield(f)
    if f=1 then
        x=240
        y=150
    elseif f=2 then
        x=230
        y=150
    endif
end sub

'''
''' process the keyboard
'''
sub get_keys
    dim tmp as int
    dim cc as int

    if key(#KEY_UP) then   'move the lemming up
        tmp=y-3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_up
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_DOWN) then 'move the lemming down
        tmp=y+3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_down
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_LEFT) then 'move the lemming left
        tmp=x-3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_left
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_RIGHT) then 'move the lemming right
```

```
        tmp=x+3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_right
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
end sub

'''
''' get a string from the file
''' a string ends in CR/LF
'''
sub getstring(f as int, s[] as char)
    dim x as char
    dim n as int

    n=0

    while(true)
        get f,,x
        if x<>13 then
            if x=10 then exit while
            s[n] = x
            s[n+1] = 0
            n=n+1
        endif
    wend
end sub

'''
''' read the playfield file defined by
''' 'field' and set up the sprites and
''' the collision table
'''
sub init_playfield(field as int)
    dim fname[32] as char
    dim s[32] as char

    ''' create the filename to read
    sprint fname, "playfield", field, ".dat"

    ''' open the playfield definition file
    open fname for read as 1

    getstring(1, s) 'get the sprite name

    '''
    ''' the playfield sprite has 9 bitmaps
    ''' one for each of the different areas of the
    ''' playfield
    sprite sprite_field, s

    '''
    ''' read the 30 playfield collision rows
    '''
    for i=0 to 29
```

```
        getstring(1, collision_table[i])
    next

    '''
    ''' close the playfield.dat file
    '''
    close 1
end sub

'''
''' main subroutine
'''
sub mainsub
    init
    init_playfield(1)

    while true
        inline _escape(0); /* check the escape key */
        repaint
        get_keys
    wend
end sub

'''
''' call the main subroutine
'''
mainsub
```

The Variables

Two new variables are introduced here. collision_pf0 and collision_pf9 are created and initialized as '0' and '9' respectively. These variable represent the collision table entries 0,1,2,3,4,5,6,7,8,9 which in turn represent playfields 0-9. In our program only playfields 1 and 2 are used.

sub update_file

no changes here.

sub init

Here we initialize the collision_pf0 and collision_pf9 variables. We also call _load() on the playfield2.dat file. This will copy playfield2.dat to the flash memory where it can be read.

sub repaint

there were no changes to this function.

sub update_z_cnt

No changes here.

function collision_check

No changes here.

sub transport

This new function takes as a parameter the number (0-9) of the playfield to transport to. The init_playfield() function is called to perform the transportation and then the x, y variables are set to the correct playfield coordinate.

sub get_keys

get_keys() gets a major upgrade. Here we do everything we did before to determine if the lemming is allowed to enter a region. And, we check to see if the region it is entering is a portal to another playfield. If we are entering a portal, the transport() function is called so that game play may continue there.

sub getstring

no changes here

sub init_playfield

no changes here

sub mainsub

no changes here

Chapter 27: Playfield Graphics Part VI: Static Sprites

Part-4 demonstrated how to put the collision data into a file. In Part-5 we implemented multiple playfields to extend the game play. Now, in Part-6, we add static (or stationary) sprites. These are objects that the lemming can "pick up" and do not move.

Money & Points

In this example, the lemming has a goal. The goal is to collect all the money on the 2 screens. There is a money sprite at each intersection of the roads in the lemming's world. Each pile of money is worth 100 dollars and when the lemming collects the money, a sound will be emitted. Naturally, when the money is collected, it must disappear from view, giving the illusion that the lemming removed it from the world-view.

Types

Part-6 introduces a B2C Language feature you may not have encountered before. It is called a "Type". A Type is a user-supplied collection of variables. In our example, we want to supply each sprite with an x,y coordinate, a playfield it belongs to, a screen within the playfield, a sound, a dollar value, and a sprite number.

Under other conditions we might have to create separate variables for each of these items, and collect them together in arrays – one for each stationary sprite. But with Types we collect them into one large variable… like this:

```
type obj
    mysprite as int
    x as int
    y as int
    screen as int
    amt as int
    sound[32] as char
end type
```

Here we have created a new user type called "Obj" (short for object). Mysprite is the name of the entry for the sprite number of the money. X, and y are the entries for the position on the screen to position the sprite. Screen is the number of the screen (0-9) which this sprite belongs on. Amt is the dollar amount for the sprite and sound is the sound to play whenever the sprite is encountered.

To use this practically, we have to create an array of these types. This is done simply:

```
dim objs[2,10] as obj
```

Now we have 20 of these objects to use as we wish (an array of 2 x 10). The first dimension is the playfield the sprite belongs on (0,1), the second is for which sprite (0-9).

Accessing the elements is done with the "dot" operator – ".". You first select which object you want with the array designator [i,j] then use the dot operator to select the Type element. For example, to get the sprite's sound you would use: objs[1,3].sound

The Program

```
'''
''' playfield tutorial - part 6
''' stationary sprites
'''
''' by greg smith
'''
```

```
'''
''' use c coordinates
'''
OPTION C_COORDS

'''
''' turn off escape processing for speed
'''
OPTION ESCAPE OFF

dim x as int  'position of lemming from 0-480
dim y as int  'position of lemming from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming

dim sprite_field as int 'the sprite number of the playfield sprites
dim sprite_lemming as int   'the sprite number of the lemming sprites
dim lemming_left as int     'index into lemming sprite for going left
dim lemming_right as int    'index into lemming sprite for going right
dim lemming_up as int       'index into lemming sprite for going up
dim lemming_down as int     'index into lemming sprite for going down

dim collision_street as char     'the street's collision value
dim collision_pf0 as char        'playfield 0
dim collision_pf9 as char        'playfield 9

dim collision_table[30,31] as char 'the collision table

dim sprite_ctr as int        'the next sprite to be allocated
dim current_playfield as int 'the current playfield onscreen
dim points as long           'your score
dim n_playfields as int      'the number of playfields

type obj                     'the sprite object
    mysprite as int          'my sprite number
    x as int                 'x coord on screen (0-159)
    y as int                 'y coord on screen (0-99)
    screen as int            'screen within playfield (0-9)
    amt as int               'points for catching this sprite
    sound[32] as char        'the sound to make
end type

dim objs[2,10] as obj        'all the sprites

'''
''' setup object
'''
'''
''' update file - copy file from app to flash
'''
sub update_file(fname[] as char)
    inline File_remove(fname);
    inline _load(fname);
end sub

'''
''' initialization
'''
sub init
    update_file("playfield1.dat") 'get the .dat file into flash
```

```
    update_file("playfield2.dat")

    sprite_field = 0       'playfield sprite
    sprite_lemming = 31    'lemming sprite is largest possible sprite -
on top

    lemming_left=0        'lemming left image
    lemming_right=5       'lemming right image
    lemming_up=10         'lemming up image
    lemming_down=15       'lemming down image

    '''
    ''' the lemming has 20 bitmaps 5 for each direction
    sprite sprite_lemming, "lemming.pic"

    '''
    ''' the lemming can be anywhere on the playfield from
    ''' x = 0-480
    ''' y = 0-300
    x=240  'center of the playfield
    y=150  'center of the playfield
    z_dir=lemming_right  ' lemming faces right
    z_cnt=0              ' animation counter

    '''
    ''' set the collision flags
    '''
    collision_street = \x\
    collision_pf0 = \0\
    collision_pf9 = \9\

    sprite_ctr=1    'initialize the sprite counter
    obj_init_all    'init all objects
    points = 0      'reset points

    n_playfields = 2   'maximum number of playfields
end sub

sub repaint
    dim field_x as int   '0,1,2 for the horizontal bitmap
    dim field_y as int   '0,1,2 for the vertical bitmap
    dim field as int     '0-8 for the bitmap currently visible
    dim lemming_x as int     '0-159 for the x coord on the screen
    dim lemming_y as int     '0-99  for the y coord on the screen
    dim obj_x as int        ' position of the object
    dim obj_y as int        ' position of the object
    dim obj_field_x as int
    dim obj_field_y as int
    dim obj_field as int
    dim i as int

    field_x = x / 160    ' compute the horizontal bitmap
    field_y = y / 100    ' compute the vertical bitmap
    field = field_y*3 + field_x ' combine them to compute the bitmap
number

    lemming_x = x mod 160   ' compute the lemming's position on the
screen
    lemming_y = y mod 100   ' compute the lemming's position on the
screen
```

```
    move sprite_field, 0, 0, field  'display the current field
    move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
lemming

    paper white  'transparent color for the lemming
    ink ltgrey

    '''
    ''' iterate across all sprites and either display or hide them
    '''
    for i=0 to n_playfields-1
      for j=0 to 9 'objects
        '''
        ''' if the sprite is on the current playfield
        ''' and the sprite is not hidden (x=-1) then
        ''' if the sprite is on the current screen then display it
        if i=current_playfield and objs[i,j].x >= 0 then
            if objs[i,j].screen = field then ' is the sprite on-screen?
                move objs[i,j].mysprite, objs[i,j].x, objs[i,j].y
            else
                move objs[i,j].mysprite, 160, 100 'hide sprite
            endif
        else
            move objs[i,j].mysprite, 160, 100 'hide sprite
        endif
      next 'j
    next 'i

    '''
    ''' display points
    '''
    ink white
    font "mini_bold_font"
    printxy 0,0,points

    redraw        'redraw all the sprites
end sub

'''
''' increment the z_cnt variable
'''
sub update_z_cnt
    z_cnt=(z_cnt+1) MOD 5
end sub

'''
''' check for collision against table
'''
function collision_check(x as int, y as int) as char
    collision_check = collision_table[y/10,x/16]
end function

'''
''' transport lemming to another playfield
'''
sub transport(f as int)
    init_playfield(f)
    if f=1 then
        x=240
        y=150
    elseif f=2 then
```

```
        x=230
        y=150
    endif
end sub

'''
''' process the keyboard
'''
sub get_keys
    dim tmp as int
    dim cc as int

    if key(#KEY_UP) then   'move the lemming up
        tmp=y-3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_up
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_DOWN) then 'move the lemming down
        tmp=y+3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_down
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_LEFT) then 'move the lemming left
        tmp=x-3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_left
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_RIGHT) then 'move the lemming right
        tmp=x+3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_right
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
end sub
```

```
'''
''' get a string from the file
''' a string ends in CR/LF
'''
sub getstring(f as int, s[] as char)
    dim x as char
    dim n as int

    n=0

    while(true)
        get f,,x
        if x<>13 then
            if x=10 then exit while
            s[n] = x
            s[n+1] = 0
            n=n+1
        endif
    wend
end sub

sub obj_init(pfld as int, objno as int, fname[] as char, x as int, y as
int, amt as int, sound[] as char)
    objs[pfld,objno].x = x mod 160
    objs[pfld,objno].y = y mod 100
    objs[pfld,objno].screen = (x/160)+(y/100)*3
    objs[pfld,objno].mysprite = sprite_ctr
    objs[pfld,objno].amt = amt
    objs[pfld,objno].sound = sound
    sprite sprite_ctr, fname
    move sprite_ctr, 160, 100 'hide sprite
    sprite_ctr = sprite_ctr+1
end sub

'''
''' init all objects
'''
sub obj_init_all

    print "Loading Sprites..."

    obj_init(0, 0, "money.pic", 32,50,100, "money.mus")
    obj_init(0, 1, "money.pic", 20,140,100, "money.mus")
    obj_init(0, 2, "money.pic", 20,240,100, "money.mus")
    obj_init(0, 3, "money.pic", 224,40,100, "money.mus")
    obj_init(0, 4, "money.pic", 224,140,100, "money.mus")
    obj_init(0, 5, "money.pic", 224,240,100, "money.mus")
    obj_init(0, 6, "money.pic", 432,40,100, "money.mus")
    obj_init(0, 7, "money.pic", 432,110,100, "money.mus")
    obj_init(0, 8, "money.pic", 432,170,100, "money.mus")
    obj_init(0, 9, "money.pic", 432,240,100, "money.mus")

    print "Still Loading Sprites..."

    obj_init(1, 0, "money.pic", -1, -1, 100, "money.mus")
    obj_init(1, 1, "money.pic", 64, 40, 100, "money.mus")
    obj_init(1, 2, "money.pic", 64,140, 100, "money.mus")
    obj_init(1, 3, "money.pic", 64,260, 100, "money.mus")
    obj_init(1, 4, "money.pic",224, 40, 100, "money.mus")
    obj_init(1, 5, "money.pic",224,140, 100, "money.mus")
```

```
    obj_init(1, 6, "money.pic",224,260, 100, "money.mus")
    obj_init(1, 7, "money.pic",384, 40, 100, "money.mus")
    obj_init(1, 8, "money.pic",384,140, 100, "money.mus")
    obj_init(1, 9, "money.pic",384,260, 100, "money.mus")
end sub

'''
''' read the playfield file defined by
''' 'field' and set up the sprites and
''' the collision table
'''
sub init_playfield(field as int)
    dim fname[32] as char
    dim s[32] as char

    current_playfield = field-1

    ''' create the filename to read
    sprint fname, "playfield", field, ".dat"

    ''' open the playfield definition file
    open fname for read as 1

    getstring(1, s) 'get the sprite name

    '''
    ''' the playfield sprite has 9 bitmaps
    ''' one for each of the different areas of the
    ''' playfield
    sprite sprite_field, s

    '''
    ''' read the 30 playfield collision rows
    '''
    for i=0 to 29
        getstring(1, collision_table[i])
    next

    '''
    ''' close the playfield.dat file
    '''
    close 1
end sub

'''
''' check collisions
'''
sub check_collisions
    dim i as int
    dim j as int

    i = current_playfield
    for j=0 to 9
        if collision(sprite_lemming, objs[i,j].mysprite) then
            objs[i,j].x=-1 ' turn off this sprite
            points = points + objs[i,j].amt
            music foreground, objs[i,j].sound
            music foreground, play
        end if
    next
end sub
```

```
'''
''' main subroutine
'''
sub mainsub
    init
    init_playfield(1)

    while true
        inline _escape(0); /* check the escape key */
        repaint
        get_keys
        check_collisions
    wend
end sub

'''
''' call the main subroutine
'''
mainsub
```

New Variables

Several new variables are introduced including a new data type. Sprite_ctr keeps track of the last sprite we used. Current_playfield is the currently selected playfield. Points is introduced to keep track of the number of points we've accumulated. And n_playfields tells us how many playfields we will be playing on.

We've already reviewed the use of the Type command and the new type Obj. Objs keeps track of all the objects on the screen.

sub update_file

no changes here.

sub init

sprite_ctr is initialized to 1 and obj_init_all is called to initialize all the sprites. The points variable is initialized to zero. We set the n_playfields to 2.

sub repaint

A large loop was added here to display the dollar-sprites. It is a doubly-nested loop. The 'I' variable goes from 0 to n_playfields-1 (0,1). This selects the sprite by playfield. The inner loop (j) goes from 0 to 9 iterating across each sprite in the playfield.

If the sprite is in the current playfield (I=current_playfield) and its not off screen (its x value is not negative – we set the x to –1 when the money is picked up), then if the screen the sprite is on is the currently displayed screen, then position the sprite on-screen. Otherwise the sprite is positioned off-screen (x=160 and y=100).

We also display the points on the screen for the first time. This version of the tutorial will have a "blinking" points value. This is because the points are on-screen for a short period of time before the "redraw" command clobbers it by redrawing all the sprites. We'll cover this problem in the next tutorial.

sub update_z_cnt

No changes here.

function collision_check

No changes here.

sub transport

No Changes here.

sub get_keys

No Changes Here.

sub getstring

no changes here

sub obj_init

This subroutine will take playfield coordinates for the sprite and initialize the variables of the object type array. The X and Y variables are in "playfield coordinates" which range from (0,0) to (479,299). With

them we compute the screen the sprite belongs on and the screen coordinates (0,0) to (159,99). This is the same mathematics used to position the lemming.
The my_sprite element of the object is set to sprite_ctr and sprite_ctr is incremented. The sprite is loaded with the "fname" variable and we initially hide the sprite by moving it to 160,100 (off-screen coords). The sound of the sprite is also stored.

sub obj_init_all

Here we initialize all the dollar-sprites by calling obj_init. Notice that one of the objects is positioned at "-1,-1". This is because we don't want it on the screen at all. As I mentioned earlier (in the repaint sub) an x coordinate of –1 indicates the object has been collected.

sub init_playfield

no changes here

sub check_collisions

This subroutine checks for collisions between the lemming and the dollar-sprites. For each sprite in the current playfield we check to see if there is a collision between the lemming and the dollar-sprite. If so, then we increment the points scored, play the music, and effectively "delete" the sprite by setting the X element to –1.

sub mainsub

no changes here

Chapter 28: Playfield Graphics Part VII: Sprite Reuse

Part-7 is perhaps the most difficult of this series – because we don't really add anything new to the program except the method used to display sprites. Up until now we have let the graphics engine display all the sprites. We gave it the X/Y coordinates and said "redraw" and everything came out lovely.

But as the program gets more advanced we need more control over the way the sprites are displayed. Did you wonder to yourself during Part-6 "Why are we using the same sprite 19 times for the money? Wouldn't it be more efficient to use the same sprite over and over again?" And of course the answer is "Yes".

Double Buffering

In this part of the tutorial, we'll load the "money.pic" sprite once into Sprite 1 and use it to "stamp" pictures of the money on each of the screens. To accomplish this we have to turn automatic "Show" off.

```
OPTION SHOW OFF
```

Normally, when we use any of the graphic functions like printxy, or redraw, or line, the result of the command is immediately apparent on the screen. That is because an internal drawing function called "Show" is executed right away. You see, all graphics commands are actually drawn on a backing display called a buffer. The buffer is drawn on until a "Show" is performed. Then it becomes visible to the user. This technique of buffering graphics is called "double-buffering".

Because B2C was created for the first time programmer, this Show was performed after each graphics command. The problem is it's very inefficient. It makes more sense to draw everything in the buffer then do a Show at the end (rather than a Show after every command). This is why so many B2C programs are both slow and display lots of flicker.

Sprite Reuse

To use the same sprite over and over again, we have to first load the sprite into memory. Then we move the sprite to the first location, call the redraw function for that sprite (and that sprite only). Then move the sprite to the next location and redraw it. And so on. Each time we redraw the sprite it acts like a "stamp" imprinting an image of the sprite on the buffered display. Finally, when all the sprites have been imprinted, we call "Redraw Show" and the buffer is copied to the main display for the user to see. Here is a sample :

```
Sprite 1, "money.pic"
CLS
Move 1, 0, 0 'upper left of screen
Redraw 1 ' imprint the money on the buffered display
Move 1, 50, 50 'middle of screen
Redraw 1 'imprint the money on the buffered display
Move 1, 100, 75 'lower right of screen
Redraw 1 'imprint the money on the buffered display
Redraw Show ' show the buffer to the user "Show me the Money!"
```

What We're Doing

In this part of the program we are actually gearing up for a fully data-driven program. Instead of loading the same sprite over and over again, we create a little list of names of sprites we've already loaded. A function (Load_sprite) is created which takes a sprite's filename and returns a sprite number. If the sprite's filename is not in our list, we add it to the list and load the sprite and return the sprite number. If the sprite's filename IS in our list we just return the sprite number.

We are also modifying the repaint logic to use sprites over and over again. And we change the collision logic. I'll explain those in more detail below.

The Program

```
'''
''' playfield tutorial - part 7
''' sprite sharing
'''
''' by greg smith
'''

'''
''' use c coordinates
'''
OPTION C_COORDS

'''
''' turn off escape processing for speed
'''
OPTION ESCAPE OFF

'''
''' turn off automatic graphics show
'''
OPTION SHOW OFF

dim x as int  'position of lemming from 0-480
dim y as int  'position of lemming from 0-300
dim z_dir as int  'which direction of lemming
dim z_cnt as int  'animation image of lemming

dim sprite_field as int 'the sprite number of the playfield sprites
dim sprite_lemming as int   'the sprite number of the lemming sprites
dim lemming_left as int     'index into lemming sprite for going left
dim lemming_right as int    'index into lemming sprite for going right
dim lemming_up as int       'index into lemming sprite for going up
dim lemming_down as int     'index into lemming sprite for going down

dim collision_street as char     'the street's collision value
dim collision_pf0 as char        'playfield 0
dim collision_pf9 as char        'playfield 9

dim collision_table[30,31] as char 'the collision table

dim obj_ctr as int        'the next object to be allocated
dim current_playfield as int 'the current playfield onscreen
dim current_screen as int
dim points as long           'your score
dim n_playfields as int      'the number of playfields
dim current_sprite as int    'the number of loaded sprites

type obj                     'the sprite object
    mysprite as int          'my sprite number
    x as int                 'x coord on screen (0-159)
    y as int                 'y coord on screen (0-99)
    playfield as int         'playfield this sprite belongs on
    screen as int            'screen within playfield (0-9)
    amt as int               'points for catching this sprite
    sound[32] as char        'the sound to make
end type
```

```
dim objs[40] as obj        'all the sprites
dim sprite_names[30,32] as char ' sprite names

'''
''' update file - copy file from app to flash
'''
sub update_file(fname[] as char)
    inline File_remove(fname);
    inline _load(fname);
end sub

'''
''' initialization
'''
sub init
    update_file("playfield1.dat") 'get the .dat file into flash
    update_file("playfield2.dat")

    sprite_field = 0       'playfield sprite
    sprite_lemming = 31    'lemming sprite is largest possible sprite -
on top

    lemming_left=0        'lemming left image
    lemming_right=5       'lemming right image
    lemming_up=10         'lemming up image
    lemming_down=15       'lemming down image

    '''
    ''' the lemming has 20 bitmaps 5 for each direction
    sprite sprite_lemming, "lemming.pic"

    '''
    ''' the lemming can be anywhere on the playfield from
    ''' x = 0-480
    ''' y = 0-300
    x=240  'center of the playfield
    y=150  'center of the playfield
    z_dir=lemming_right  ' lemming faces right
    z_cnt=0              ' animation counter

    '''
    ''' set the collision flags
    '''
    collision_street = \x\
    collision_pf0 = \0\
    collision_pf9 = \9\

    obj_ctr=0    'initialize the object counter
    points = 0      'reset points

    n_playfields = 2   'maximum number of playfields
    current_sprite = 1 'the next sprite number available

    obj_init_all    'init all objects

end sub

sub repaint
    dim field_x as int   '0,1,2 for the horizontal bitmap
    dim field_y as int   '0,1,2 for the vertical bitmap
```

```
    dim lemming_x as int     '0-159 for the x coord on the screen
    dim lemming_y as int     '0-99  for the y coord on the screen
    dim obj_x as int        ' position of the object
    dim obj_y as int        ' position of the object
    dim obj_field_x as int
    dim obj_field_y as int
    dim obj_field as int
    dim i as int

    field_x = x / 160    ' compute the horizontal bitmap
    field_y = y / 100    ' compute the vertical bitmap
    current_screen = field_y*3 + field_x ' combine them to compute the
bitmap number

    lemming_x = x mod 160   ' compute the lemming's position on the
screen
    lemming_y = y mod 100   ' compute the lemming's position on the
screen

    move sprite_field, 0, 0, current_screen 'display the current field
    redraw sprite_field

    paper white  'transparent color for the lemming
    ink ltgrey

    '''
    ''' iterate across all sprites and display
    '''
    for i=0 to obj_ctr-1
        '''
        ''' if the sprite is on the current playfield
        ''' and the sprite is not hidden (x=-1) then
        ''' if the sprite is on the current screen then display it
        '''
        if objs[i].playfield=current_playfield and objs[i].x >= 0 then
            if objs[i].screen = current_screen then ' is the sprite on-
screen?
                move objs[i].mysprite, objs[i].x, objs[i].y
                redraw objs[i].mysprite
             endif
        endif
    next 'i

    '''
    ''' display points
    '''
    ink white
    font "mini_bold_font"
    printxy 0,0,points

    move sprite_lemming, lemming_x, lemming_y, z_dir+z_cnt 'display the
lemming
    redraw sprite_lemming

    redraw show        'redraw all the sprites
end sub

'''
''' increment the z_cnt variable
'''
sub update_z_cnt
```

```
    z_cnt=(z_cnt+1) MOD 5
end sub

'''
''' check for collision against table
'''
function collision_check(x as int, y as int) as char
    collision_check = collision_table[y/10,x/16]
end function

'''
''' transport lemming to another playfield
'''
sub transport(f as int)
    init_playfield(f)
    if f=1 then
        x=240
        y=150
    elseif f=2 then
        x=230
        y=150
    endif
end sub

'''
''' process the keyboard
'''
sub get_keys
    dim tmp as int
    dim cc as int

    if key(#KEY_UP) then   'move the lemming up
        tmp=y-3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_up
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_DOWN) then 'move the lemming down
        tmp=y+3
        cc = collision_check(x,tmp)
        if cc = collision_street then
            y=tmp
            z_dir=lemming_down
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_LEFT) then 'move the lemming left
        tmp=x-3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_left
```

```
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
    if key(#KEY_RIGHT) then 'move the lemming right
        tmp=x+3
        cc = collision_check(tmp,y)
        if cc = collision_street then
            x=tmp
            z_dir=lemming_right
            update_z_cnt
        end if
        if cc >= collision_pf0 and cc <= collision_pf9 then
            transport(cc-collision_pf0)
        end if
    end if
end sub

'''
''' get a string from the file
''' a string ends in CR/LF
'''
sub getstring(f as int, s[] as char)
    dim x as char
    dim n as int

    n=0

    while(true)
        get f,,x
        if x<>13 then
            if x=10 then exit while
            s[n] = x
            s[n+1] = 0
            n=n+1
        endif
    wend
end sub

'''
''' load_sprite will find a previously loaded sprite and return its
''' sprite number or load the sprite and add its name to the list
''' and return the new sprites index
'''
function load_sprite(name[] as char)
    dim found as int
    found = false
    for i=0 to current_sprite-1
        if name = sprite_names[i] then
            found = true
            load_sprite = i+1
            exit for
        end if
    next 'i
    if not found then
        load_sprite = current_sprite
        sprite current_sprite, name
        sprite_names[current_sprite-1]=name
        current_sprite = current_sprite + 1
```

```
    end if
end function

'''
''' obj_init will allocate a single object and attach it to a sprite
'''
sub obj_init(pfld as int, fname[] as char, x as int, y as int, amt as
int, sound[] as char)
    objs[obj_ctr].playfield = pfld
    objs[obj_ctr].x = x mod 160
    objs[obj_ctr].y = y mod 100
    objs[obj_ctr].screen = (x/160)+(y/100)*3
    objs[obj_ctr].mysprite = load_sprite(fname)
    objs[obj_ctr].amt = amt
    objs[obj_ctr].sound = sound
'   sprite command deleted
'   move obj_ctr, 160, 100 'hide sprite
    obj_ctr = obj_ctr+1
end sub

'''
''' init all objects
'''
sub obj_init_all

    print "Loading Sprites..."

    obj_init(0, "money.pic", 32,50,100, "money.mus")
    obj_init(0, "money.pic", 20,140,100, "money.mus")
    obj_init(0, "money.pic", 20,240,100, "money.mus")
    obj_init(0, "money.pic", 224,40,100, "money.mus")
    obj_init(0, "money.pic", 224,140,100, "money.mus")
    obj_init(0, "money.pic", 224,240,100, "money.mus")
    obj_init(0, "money.pic", 432,40,100, "money.mus")
    obj_init(0, "money.pic", 432,110,100, "money.mus")
    obj_init(0, "money.pic", 432,170,100, "money.mus")
    obj_init(0, "money.pic", 432,240,100, "money.mus")

    print "Still Loading Sprites..."

    obj_init(1, "money.pic", 64, 40, 100, "money.mus")
    obj_init(1, "money.pic", 64,140, 100, "money.mus")
    obj_init(1, "money.pic", 64,260, 100, "money.mus")
    obj_init(1, "money.pic",224, 40, 100, "money.mus")
    obj_init(1, "money.pic",224,140, 100, "money.mus")
    obj_init(1, "money.pic",224,260, 100, "money.mus")
    obj_init(1, "money.pic",384, 40, 100, "money.mus")
    obj_init(1, "money.pic",384,140, 100, "money.mus")
    obj_init(1, "money.pic",384,260, 100, "money.mus")
end sub

'''
''' read the playfield file defined by
''' 'field' and set up the sprites and
''' the collision table
'''
sub init_playfield(field as int)
    dim fname[32] as char
    dim s[32] as char

    current_playfield = field-1
```

```
    ''' create the filename to read
    sprint fname, "playfield", field, ".dat"

    ''' open the playfield definition file
    open fname for read as 1

    getstring(1, s) 'get the sprite name

    '''
    ''' the playfield sprite has 9 bitmaps
    ''' one for each of the different areas of the
    ''' playfield
    sprite sprite_field, s

    '''
    ''' read the 30 playfield collision rows
    '''
    for i=0 to 29
        getstring(1, collision_table[i])
    next

    '''
    ''' close the playfield.dat file
    '''
    close 1
end sub

'''
''' check collisions
'''
sub check_collisions
    for i=0 to obj_ctr-1
      if  objs[i].playfield  =  current_playfield  and  objs[i].screen  =
current_screen and objs[i].x>=0 then
        move objs[i].mysprite, objs[i].x, objs[i].y
        if collision(sprite_lemming, objs[i].mysprite) then
            objs[i].x=-1 ' turn off this sprite
            points = points + objs[i].amt
            music foreground, objs[i].sound
            music foreground, play
            exit for
        end if
      endif
    next
end sub

'''
''' main subroutine
'''
sub mainsub
    init
    init_playfield(1)

    while true
        inline _escape(0); /* check the escape key */
        repaint
        get_keys
        check_collisions
    wend
end sub
```

```
'''
''' call the main subroutine
'''
mainsub
```

New Variables

Current_screen is added, it is the number of the screen currently on display.

Obj_ctr is the number of objects loaded so far. This is not to be confused with sprites. An object (see Type Obj command) is something that appears on the screen. It has sound, and position, and point value (amt) and a playfield and screen it belongs to. But most importantly it has a sprite number (mysprite in the type declaration).

We increased the objs[] array to 40 elements, though we still only use 19. This is the list of objects in the game. It used to be a 2-dimensional array, but for future purposes makes more sense as a single list.

N_playfields is the number of playfields in our game (2 for us).
Current_sprite is the number of sprites we have loaded into memory. Sprite_names[] is the list of filenames of sprites we have loaded.

sub update_file

no changes here.

sub init

We reset the obj_ctr, set the n_playfields to 2 and the current_sprite to 1 (because sprite 0 is the background - Sprite 1 is the first sprite we can load into memory). A call to Obj_init_all was moved to the bottom of the function because it depends on the current_sprite variable being initialized.

sub repaint

This function got the most attention in this part of the tutorial. We changed the "field" variable to the global "current_screen" because a) it's a better name anyway and b) we need it later for collision detection.

The first thing we do is draw the background sprite.

```
move sprite_field, 0, 0, current_screen 'display the current field
redraw sprite_field
```

Notice that we called "redraw sprite_field" this stamps the image of the background screen over the entire display. No need to call CLS because everything is covered up by this operation. So we save a couple milliseconds by not calling CLS, heh.

Next we iterate over all the objects in the program. This is different than the last program where we iterated across a 2-dimensional array. Here we iterate across all objects defined and check to see if each object is a) on the current playfield b) enabled (x>=0) and c) on the current_screen. If so then we call Move followed by Redraw. Note that we use whatever "mysprite" points to. In this program it will always be the money.pic sprite. Ahhh… reuse at it's best.

Finally we printxy the points and Move/Redraw the lemming. And as the last thing to do – Redraw Show to display the buffer to the display screen.

You may notice slightly better gameplay and less flicker in this version of the game.

sub update_z_cnt

No changes here.

function collision_check

No changes here.

sub transport

No Changes here.

sub get_keys

No Changes Here.

sub getstring

no changes here

function load_sprite

This new function is called to load a sprite if it hasn't already been loaded, and return the value of a sprite if it already has been. The logic should be pretty easy to follow.

sub obj_init

This subroutine changed a little since I changed the objs[] array to a single dimension. The obj_ctr points to the current object to create. We set the playfield to the pfld value passed in. We compute the x/y/screen for the coordinates passed in and call load_sprite to get the sprite number. We also took out the sprite command and the move command since they are handled in our new redraw subroutine.

sub obj_init_all

Again, this subroutine changed a little bit because we no longer have the 2-d array of objs.

sub init_playfield

no changes here

sub check_collisions

This subroutine changed substantially because the position of the sprites is no longer stored in the sprite itself but in the objs[] array. So for each object we must move the sprite into position and call the collision() function to see if they overlap. If they do overlap we play music and increment points.

sub mainsub

no changes here

Chapter 29: B2Cbuild

B2Cbuild is a process that will read a control file (filename.bld) and create all intermediate files for the Cybiko SDK. The format of the .bld file is similar to the Windows .ini file: a bracketed command followed by lines of parameters.

Example:
[source]
tardis.b2c

The single command-line argument to b2cbuild is either a .bld file or a .b2c file to build. If you specify a .b2c file then all defaults are taken and a vanilla application is created. If the .bld file is specified, it is opened and read. In this case a [source] command is required.

The [source] command is required. All other commands are optional. Order is not important.

Comments are delimited by a '#' (pound-sign) in column one of the line. Blank lines are also considered comments.

[3dsprite=filename.spr]

The 3dsprite command creates sprites for 3d gaming. It requires a parameter which is the name of the .spr file to create. The 3dsprite command is followed by one or more lines of text each of which is a bitmap (.bmp) to be converted to the .spr format and included within the filename.spr file. Each bitmap must be 32x32 pixels and 4 color greyscale. Multiple files can be specified allowing for animated sprites.
Default: none
Example:

```
[3dsprite=filename.sprite]
lemming01.bmp
lemming02.bmp
lemming03.bmp
lemming04.bmp
lemming05.bmp
```

[3dtex]

The 3dtex command creates .tex (texture) files for 3d gaming. It is followed by a single line with the name of a bitmap file. The bitmap may be either 32x32 pixels (for a full tile) or 32x16 pixels (for a half tile). In either case the bitmap must be 4-color greyscale.
Default: none
Example:

```
[3dtex]
wall.bmp
```

[author]

A single line with the author's name follows the author command. This will appear in the Splash page when the application first shows up.
Default: none
Example:

```
[author]
Gregory Smith
```

[copyright]

The copyright command is followed by a single line with the copyright line of the program. The copyright appears in the Splash text at the beginning of the program.
Default: (C) Copyright 2001 (*whatever the current year is*)
Example:

```
[copyright]
©2001, The Alcor Group
```

[cyos]

The cyos command is followed by a single line with the minimum CyOS version that is needed to run the application.
Default:

For [sdk] > 2 : cyos=1.5.1
For other sdk : cyos=1.3.57

Example:

```
[cyos]
1.5.1
```

[environment]

The environment command is followed by one or more lines setting environment variables in the form of name=value. There are 3 special environment variables. CYBIKO_SDK is the path to where you installed the Cybiko SDK. PATH is the list of directories where executables can be found. And CC is the C compiler (normally vcc). You can modify CC to include optimization options.
Default: CC=vcc
Example:

```
[environment]
CYBIKO_SDK=C:\program files\cybiko\sdk3.0.pro
PATH=C:\program files\cybiko\sdk3.0.pro\bin;%PATH%
CC=vcc –V15
```

[files]

The files command is followed by one or more lines of text each of which is a file name which will be used by the application.
Default: none
Example:

```
[files]
datafile1.dat
rooms.dat
```

[help]

The help command is followed by one or more lines of helpful text. This text will appear in the help window when the user hits the "?" help key in the application.
Default: none
Example:

```
[help]
Tardis Adventure
This is the help text for tardis
N=north
S=south
E=east
W=west
```

[icon]

The icon command is followed be a single line of text which is the filename of a bitmap (.bmp) file to be converted into root.ico file for use as the application's on-screen icon. (48x47 pixels)
Default: inc\b2c.bmp
Example:

```
[icon]
tardis.bmp
```

[icon0]

The icon0 command is followed be a single line of text which is the filename of a bitmap (.bmp) file to be converted into 0.ico file for use as the application's on-screen icon (SDK 3 only, 30x18 pixels)
Default: inc\0.bmp
Example:

```
[icon0]
tardis.bmp
```

[icon1]

The icon1 command is followed be a single line of text which is the filename of a bitmap (.bmp) file to be converted into 1.ico file for use as the application's menu icon (SDK 3 only, 9x9 pixels)
Default: inc\1.bmp
Example:

```
[icon1]
tardis.bmp
```

[include=filename.bld]

The include command suspends reading the current file and starts reading the include file. Processing will continue in the original file when the included file is completely read.
Default: none
Example:

```
[include=sdk3.bld]
```

[music]

The music command is followed by one or more lines of text each of which is a music file (.mus) which will be used by the application.
Default : none
Example :

```
[music]
tardis.mus
tada.mus
theme.mus
```

[name]

The name command is followed by a single line with the name of the application as it will appear on the Cybiko's menu.
Default: the name of the source file stripped of the .b2c extension
Example:

```
[name]
Tardis Adventure
```

[objects]

The objects command is followed by one or more lines of text, each of which is an object file previously compiled by the vcc command. The objects will be linked into the resultant program.
Default: none
Example:

```
[objects]
FileListForm.o
Jed.o
```

[option]

The option command is followed by one or more lines of text, each of which is an option to be set. There is currently only one option – "keep". If the keep option is specified then all intermediate files are left behind for user inspection. If keep is not specified, then the intermediate files are all deleted.
Default: none
Example:

```
[option]
keep
```

[output]

The output command is followed by one line of text, which is the name of the output program
Default: same as the [source] file with .app (foo.app, for example)
Example:

```
[option]
xfoo.app
```

[path]

The path command is followed by a single line with the path to the B2C directory. This path is used to find the executables b2cpp, b2c, the object file b2cuser.o, default icons, and the include file b2cuser.h.
Default: . (*current directory *)
Example:

```
[path]
c:\unzipped\b2cv3d
```

[pic=filename.pic]

The pic command requires a parameter which is the name of the .pic file to create. The pic command is followed by one or more lines of text each of which is a bitmap (.bmp) to be converted to the .pic format and included within the filename.pic file. Multiple files can be specified allowing for animated sprites.
Default: none
Example:

```
[pic=filename.pic]
lemming01.bmp
lemming02.bmp
lemming03.bmp
lemming04.bmp
lemming05.bmp
```

[sdk]

The sdk command is followed by a single line with the SDK version of your compiler. Currently only SDK of 2 or 3 is allowed.
Default: 2

Example:

```
[sdk]
3
```

[source]

The source command is followed by one or more lines with the names of the source files. This is the only required field. You may specify both .b2c and .c files. The first file is considered the "main" file and will be used to name the output file if the [output] command is not specified.

Default: none

Example:

```
[source]
tardis.b2c

or

[source]
mygame.b2c
myutils.c
moreutils.c
```

[splash.image]

The splash.image command is followed by a single line of text which is the name of a bitmap file (.bmp) which will be converted into the intro.pic file for use as the application's on-screen splash page (displayed when the app runs).

Default: none

Example:

```
[splash.image]
tardis_intro.bmp
```

[splash.text]

The splash.text command is followed by one or more lines of text to be displayed at the startup of an application. The [name], [version], [author] and [copyright] fields are displayed followed by any text entered here.

Default: none

Example:

```
[splash.text]
Welcome to the Tardis Adventure
To begin playing press any key
```

[type]

The type command is followed by a single line with the type of the program. It must be 'root', 'game' or 'app'.

Default: game

Example:

```
[type]
game
```

[version]

The version command is followed by a single line with the version string. This appears in the Splash page and in the root.inf file.

Default: 1.0.0

Example:

[version]
1.0.0

Chapter 30: **B2C Language Reference Guide**

Effectivity:

B2Cv2 : 6/1/2001
B2Cv2a : 6/24/2001
B2Cv2b : 7/6/2001
B2Cv2c : 7/13/2001
B2Cv3 : 8/11/2001
B2Cv3a : 8/26/2001
B2Cv3d : 9/25/2001
B2Cv3e : 12/12/2001
B2Cv4 : 1/1/2002
B2Cv5 : 4/21/2002

Language Caveats:

1) Graphics commands and Print commands cannot be reliably intermixed. Print commands cause a repaint of the display on each Print command. This will erase any Line, Point, or Printxy results.

*Topic : **Character-Constant***

Language Level : B2C Extension
Effectivity : B2C v3a
Syntax :

\character constant\

Description :

\ begins a character constant. A single character is placed between the backslash. This is useful in place of using the ASCII character number as in CyBasic.

Example :

```
Dim a as char
A=\x\ ‘ set a to the x character
```

*Topic : **C-Constant***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

#constant_name

Description :

identifies the name of a defined constant in the Cybiko.h file. The name which follows (usually in all uppercase) usually represents a constant which is created with the #define C compiler directive. The name is passed through to the C compiler without change.

Example :

```
If key(#KEY_UP) then y=y+1
If key(#KEY_DOWN) then y=y-1
```

Selected Key Values:

KEY_SECTION1	KEY_SECTION2	KEY_SECTION3		
KEY_SECTION4	KEY_SECTION5	KEY_SECTION6		
KEY_SECTION7				
KEY_DOWN	KEY_LEFT	KEY_UP	KEY_RIGHT	
KEY_INS	KEY_DEL	KEY_TAB	KEY_SELECT	
KEY_ENTER	KEY_BACKSPACE	KEY_HELP	KEY_SHIFT	
KEY_CONTROL	KEY_CY	KEY_SPACE	KEY_ESC	
KEY_0	KEY_1	KEY_2	KEY_3	KEY_4
KEY_5	KEY_6	KEY_7	KEY_8	KEY_9
KEY_QUOTE	KEY_COMMA	KEY_MINUS		
KEY_SLASH	KEY_SEMICOLON	KEY_BACKSLASH		

KEY_EQUAL KEY_PERIOD	KEY_OPEN_SBRACKET KEY_CLOSE_SBRACKET	KEY_BACKQUOTE		
KEY_A	KEY_B	KEY_C	KEY_D	KEY_E
KEY_F	KEY_G	KEY_H	KEY_I	KEY_J
KEY_K	KEY_L	KEY_M	KEY_N	KEY_O
KEY_P	KEY_Q	KEY_R	KEY_S	KEY_T
KEY_U KEY_Z	KEY_V	KEY_W	KEY_X	KEY_Y

*Topic : **_ptr_main_module***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

_ptr_main_module

Description :

_ptr_main_module is a constant for use in inline calls. It is a pointer to the main_module structure used by B2C

Example :

Inline DisplayGraphics_show(_ptr_main_module);

*Topic : **3dCollision***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

result = 3dCollision(sprite1, sprite2, dist)

Description :

3dCollision is a function which will return TRUE if "sprite1" is within "dist" pixels of "sprite2". This can be used to determine if two 3d sprites have run into each other. Note that sprite 0 is the camera and is a valid input to this function.

Example :

if (3dCollision(Camera, 1, 30)) then music foreground, play

*Topic : **3dGet***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

3dGet spriteno, x, y, z
3dGet Camera, x, y, dir

Description :

3dGet will retrieve the current values of the x , y, and z variables from a sprite (and the x, y, and dir variables for a Camera). This is especially useful after moving a sprite or camera with the 3dRMove command.

Example :

3dGet 1,x,y,z

*Topic : **3dMove***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

3dMove sprite_num, x,y,z
3dMove Camera, x,y,dir

Description :

3dMove will move either a sprite or the camera position. In the first form it moves a sprite number "sprite_num" to the coordinates x,y. The 'z' variable is which bitmap in a multi-bitmap sprite to display.

3dMove will position the Camera to the x,y position on the screen and set it pointing in the direction 'dir'. The 'dir' is an angle measured in degrees.

Example :

3dMove 3,0,0,1
3dMove Camera, 0,0,45

*Topic : **3dRedraw***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

3dRedraw

Description :

3dRedraw will redraw the current 3dRoom. First all walls are rendered followed by all sprites.

Example :

3dRedraw

*Topic : **3dRMove***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

3dRMove sprite_num, r,theta,z
3dRMove Camera, r,theta,0

Description :

3dRMove will move either a sprite or the camera position relative to the current position. In the first form it moves a sprite number "sprite_num" a distance 'r' in the direction 'theta' The 'z' variable is which bitmap in a multi-bitmap sprite to display. The direction 'theta' is measured in degrees.

3dRMove will position the Camera a distance 'r' relative to the current position in the direction 'theta'. The last parameter is ignored. The direction 'theta' is measured in degrees.

Example :

3dRMove 3,10,45,1
3dMove Camera, 10,45,0

*Topic : **3dRoom***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

3dRoom room_number

Description :

3dRoom defines the current room number for 3d graphics. When the room_number is set all 3d commands will default to the room specified by room_number.

Example :

3dRoom 0

*Topic : **3dSprite***

Language Level : B2C Extension
Effectivity : B2C v3d

Syntax :

3dSprite sprite_num, “filename.spr”

Description :

3dSprite will load a 32x32 sprite into memory and assign it to the current room’s sprites. The sprite_num is an index into the array of sprites and must not be zero (since sprite zero is the camera). The sprite must be converted from a bitmap using bmp2spr and will have the extension “.spr”. The sprite may have multiple bitmaps inside. The “dark grey” color is used as the transparent color.

Example :

3dSprite 1, “column.spr”

*Topic : **3dWall***

Language Level : B2C Extension

Effectivity : B2C v3d

Syntax :

3dWall x0, y0, x1, y1, “filename.tex”

Description :

3dWall will create a wall-segment along the line specified by x0, y0, x1, y1 with a texture of “filename.tex”. Only files with suffixes in “.tex” will do. These are files created from bitmaps using bmp2spr. They may have either 32x32 pixels or 32x16 pixels.

Example :

3dWall 0,0,100,100, “wall.tex”

*Topic : **Abs()***

Language Level : B2C Extension

Effectivity : B2C v3a

Syntax :

Abs(n)

Description :

Abs() take the absolute value of ‘n’ where ‘n’ is an Integer. If the value is less than zero, it returns the opposite of that value (making it positive). If the value of ‘n’ is zero or positive it simply returns the value.

Example :

X = abs(n)

Topic : Beep

Language Level : CyBasic-1

Effectivity : B2C v2

Syntax :

Beep n

Description :

Beep causes the speaker to emit a single tone. The range of valid values is from 0 (85Hz) to 67 (about 4075Hz) inclusive. If the index is negative, the tone will stop.

NOTE: This is a departure from CyBasic.

Example :

Beep 1

Topic : Char

Language Level : B2C Extension

Effectivity : B2C v3d

Syntax :

x=Char(key)

Description :

Function acts just as the key() function. Returns 0 immediately if no key is pressed and returns the keycode if a key is pressed. Does not return if the shift key is pressed but instead returns the next shifted key.

Example :

If getchar=#KEY_ENTER then exit sub

*Topic : **Circle***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Circle X, Y, R

Description :

Circle will draw a circle at location X, Y with a radius of R.

Example :

```
OPTION SHOW OFF
OPTION ESCAPE OFF

dim r as int

sub draw_circle(r as int)
    cls
    circle 0,0,r
    redraw show
end sub

draw_circle(r)

while true
  if key(#KEY_LEFT) then
    r=r+1
    draw_circle(r)
  endif
  if key(#KEY_RIGHT) then
    r=r-1
    draw_circle(r)
  endif
  if key(#KEY_SPACE) then exit while
wend
```

*Topic : **Circfill***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Circfill X,Y,R

Description :

Circfill will draw a filled circle at location X, Y with a radius of R.

Example :

```
OPTION SHOW OFF
OPTION ESCAPE OFF

dim r as int

sub draw_circle(r as int)
    cls
    circfill 0,0,r
    redraw show
```

```
end sub

draw_circle(r)

while true
  if key(#KEY_LEFT) then
    r=r+1
    draw_circle(r)
  endif
  if key(#KEY_RIGHT) then
    r=r-1
    draw_circle(r)
  endif
  if key(#KEY_SPACE) then exit while
wend
```

*Topic : **Close***

Language Level : Cybasic-1
Effectivity : B2C V2
Syntax :

close n

Description :

Closes File number N

Errors:

#ERROR_FILE_NUMBER – the file number was out of the valid range
#ERROR_FILE_ALREADY_CLOSED – the file was already closed

Example :

Close 1
if error = #ERROR_FILE_ALREADY_CLOSED then print “Ooops”

Topic : Cls

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Cls

Description :

Cls will clear the screen and paint it with the background color (see Paper).

Example :

Cls
Printf "Top of the screen"

*Topic : **Collision***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

Collision(a,b)

Description :

Collision(a,b) returns TRUE (1) if two sprites overlap and FALSE (0) otherwise. Collision detection is limited to the overlapping of the two sprite rectangles. Even if the pixels are clear, any overlap of the bounding box of the two sprites returns true.

Example

Dim x as int
Dim y as int
Dim z as int
Dim a as int

```
Dim b as int
Sprite 1, "lemming.pic"
Sprite 2, "other.pic
X=0
Y=0
Z=0
A=-80
B=-50
Move 2, a, b
While 1
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move x,y,z
        Beep 0
        Vibrate 0
        If Collision(1,2) then
                beep 1
                vibrate 128
        Redraw
wend
```

*Topic : **Cos()***

Language Level : CyBasic-1
Effectivity : B2C v5
Syntax :

Cos(n)

Description :

Cos() returns the cosine of the Double value ‘n’ (in radians).

Example :

X = cos(n)

*Topic : **Dabs()***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Dabs(n)

Description :

Dabs() take the absolute value of ‘n’ where ‘n’ is a Double variable. If the value is less than zero, it returns the opposite of that value (making it positive). If the value of ‘n’ is zero or positive it simply returns the value.

Example :

X = dabs(n)

*Topic : **Dialog***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

Result = Dialog(box_hdr, prompt)
Result = Dialog(box_hdr, prompt, str_var, len)

Description :

Dialog will display a dialog box with the OK and CANCEL buttons in it. The “box_hdr” is the text to display in the dialog box’s header while the “prompt” is the text to display in the body of the dialog box. If str_var is specified, then an Edit field appears in the dialog. Whatever is in the

str_var will be the initial value of the Edit field. “Len” is the maximum number of characters to enter into str_var (usually the dimensioned length of the string). When the user enters text into the Edit field and presses Enter, the str_var receives the input string.

Result receives TRUE if the user hits the OK button and FALSE if the user hits the CANCEL button or the ESC key.

Example :

If Dialog(“Exit?”, “Are you ready to exit now?”) then exit program
Proceed = Dialog(“Input File”, “What input file will you use?”, filename, 32)

Topic : Dim

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Dim varname{[arraysize] {As datatype}

Description :

The Dim keyword defines a variable in anticipation of its use. With the exception of For loop variables, a variable must be defined before it can be used. If the "As datatype" portion is specified, then the variable is created as the defined datatype:

Char : 1-byte signed integer or character
Int : 2-byte signed integer
Long : 4-byte signed integer
Double : 8-byte floating point (not available)

If the [arraysize] portion is included, the variable is declared as an array. B2C Supports 1 and 2 dimensions. For 2-dimensions specify the dimensions with a comma separator:

Dim a[10] as int ‘one-dimension
Dim a[10,5] as int ‘ two dimensions

You can defereference the array the same way:

A[0] = 1 ‘one dimension
A[1,2] = 1 ‘ two dimension

Example

```
Dim age as int
Dim name[32] as char
Input "age:", age
Input "name:", name
Print "You are ", name, "and your age is", age
```

Topic : eof

Language Level : Cybasic-2
Effectivity : B2C v2
Syntax :

eof(n)

Description :

Returns TRUE if the file number ‘n’ is at the end.

Errors:

#ERROR_FILE_NUMBER – the file number is outside the valid range
#ERROR_FILE_UNOPENED – the requested file has not been opened

Example :

If getchar=#KEY_ENTER then exit sub

*Topic : **Error***

Language Level : CyBasic-3

Effectivity : B2C v3e
Syntax :
error (a variable)
Description :
The Error variable receives a value during certain operations (like File I/O). It can be used to make a decision if a command fails.

#ERROR_FILE_SEEK 52
#ERROR_FILE_WRITE 53
#ERROR_FILE_READ 54
#ERROR_FILE_OPEN 56
#ERROR_FILE_CREATE 57
#ERROR_FILE_CLOSE 100
#ERROR_FILE_NUMBER 101
#ERROR_FILE_ALREADY_OPEN 102
#ERROR_FILE_ALREADY_CLOSED 103
#ERROR_FILE_UNOPENED 104
Example
```
open “myfile.txt” for READ as 0
if error <> 0 then
  print “Error opening myfile.txt”
  exit program
end if
```

Topic : ***Exists***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :
n = exists(filename)
Description :
Exists will detect if the file is present on the flash memory of the Cybiko. It returns TRUE (1) if the file is found and returns FALSE (0) otherwise.
Example :
If exists(“trash.tmp”) then remove “trash.tmp”

Topic : ***Exit-Program***

Language Level : CyBasic-3
Effectivity : B2C v3
Syntax :
Exit Program
Description :
Exit Program will terminate a B2C application immediately.
Example :
If key=#KEY_ENTER then exit program

Topic : ***False***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :
false
Description :
Constant value returns false – or zero.
Example :

If key(#KEY_ENTER)=false then beep -1

Topic : ***Filelist***

Language Level : B2C Extension
Effectivity : B2C v3e
Syntax :

rc = Filelist *(title, pattern, return_string)*

Description :

The Filelist function will display a file dialog with filenames matching pattern. Pattern is a wildcarded string using '*' as the wildcard character (for example "*.app" matches all app files). When the user highlights one of the filenames and presses ENTER the *return_string* is filled with the selected filename and TRUE is returned. If the user cancels the filelist dialog by pressing ESC then FALSE is returned.

Example :

```
Dim s[32] as char
Dim rc as int
Rc = filelist("Select App", "*.app", s)
If (rc) then print s, " Selected"
Else print "Cancelled"
```

Note : Original Dialog code by Ernest Pazera

Topic : ***Findfile***

Language Level : B2C Extension
Effectivity : B2C v3e
Syntax :

Findfile *pattern*

Description :

The Findfile command will search for files with a particular pattern. The pattern will use the wildcard character "*".

Example

```
Dim s[32] as char
Findfile "*.app"
While (nextfile(s))
        Print s
wend
```

Topic : Font

Language Level : B2C Extension
Effectivity : B2C v2a
Syntax :

Font *fontname*

Description :

Font will change the current font to the one specified by fontname (a character string or a variable containing the name of the font). You may specify a user font (in the .app archive) or any of the 4 standard fonts:

cool_normal_font
cool_bold_font (default)
mini_normal_font
mini_bold_font

Example :

```
Font "mini_normal_font"
Print "small text here"
Font "font5x7.fnt"
```

Print “larger text here”

*Topic : **For / To / Step / Exit For / Next***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

```
For varname = start-expression To end-expression {Step step-expression}
        Body
Exit For
Next
```

Description :

The For statement initiates a loop which counts from start-expression to end-expression. The varname is automatically created (without a Dim). The optional Step defines the increment to apply to the varname on each iteration. "step-expression" may be negative.

You can exit a loop early by executing the Exit For command.

Example

```
For I=1 To 100 Step 2
        Print I
        If I = 40 then Exit Program
Next I
```

*Topic : **Function***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

```
Function funcname {(param1{[]} {As datatype1} {, param2 {As datatype2} …})} {As datatype}
        {Exit Function}
        Body
End Function
```

Description :

Function initiates the definition of a new function . funcname is the name of the function. An optional parameter list may be specified. Zero, one, or more parameters may be specified. Each parameter may have a datatype as specified in Dim. However, no array parameters may be passed. If the datatype is not specified, the variable is defined as Double. Likewise, the function itself may be declared as a particular datatype, and if this datatype is unspecified the function is defined as Double.

To return a value from the function, assign a value to a variable with the same name as the function.

If the Exit Function statement is encountered, the function will return immediately.

With version 3a of B2C, single-dimensioned arrays may be passed into functions. The format of the parameter is p[] – indicating an array of arbitrary length. This now allows strings to be passed into functions and subroutines.

Example

```
Function foo as Int
        foo = random(100);
End Function

Funciton bar(div as Int) as Int
        Bar = foo /div;
End Function
```

```
Function len(s[] as char) as int
        Len=0
        For I=0 to 1000
                If s[I] = 0 then exit function
                Len = len + 1
        Next I
End Function

Print bar(2)
```

Topic : Get

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Get filenumber, {bytepos},{ variable}

Description :

Get reads a variable from a file at the current or specified byte position. The filenumber must be the number of a file already opened for Read access. If bytepos is specified, the variable will be read from that position in the file. If bytepos is unspecied, then the current position is used. If the variable is left unspecied, then the file pointer is changed to bytepos and no data is read.

Errors :

#ERROR_FILE_NUMBER – the file number is outside the valid range
#ERROR_FILE_UNOPENED – the file is not open
#ERROR_FILE_READ – the file is not readable
#ERROR_FILE_SEEK – the file could not be positioned

Example :

```
Dim foo as int
Open "a.dat" For Read as 1
Get 1, 0, foo
Print "Foo is: ", foo
Close 1
```

*Topic : **Getchar***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

x=getchar

Description :

Function waits for a single key to be pressed. Does not return if the shift key is pressed but instead returns the next shifted key. Note that unlike the key() function, this function will continue to wait until a key is pressed.

Example :

```
If getchar=#KEY_ENTER then exit sub
```

*Topic : **Getkey***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

X=getkey

Description :

Function waits for a single key to be pressed. Note that unlike the key() function, this function will continue to wait until a key is pressed. (Note: unlike the getchar function this function will return only raw keys, not shifted ones.)

Example :

```
If getkey=#KEY_ENTER then exit sub
```

Topic : Goto

Language Level : B2C Extension
Effectivity : B2C v2b
Syntax :

```
label :
Goto label
```

Description :

Transfers control from the current line to the line identified by label. It is not possible to goto a label outside the current function.

Example :

```
Dim y
Sub foo
    Dim x
    x=1
loop :
    print x,y
    x=x+1
    if x<10 then goto loop
end Subject:

y=1
count_me :
    call foo
    y=y+1
    if y<10 then goto count_me
```

*Topic : **If / Then / Else / Elseif / End If | Endif***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

```
If conditional1 Then
    Body1
{ElseIf conditional2 Then
    Body2
}
{Else
    Body3
}
End If | Endif
```

Description :

The If keyword will perform the Body1 section of code if conditional1 is true. A conditional is any numeric expression. If the conditional evaluates to zero (0) it is considered false - and the Body1 will not be executed. All other values for conditional1 are considered true and result in the Body1 being executed.

If conditional1 is false, then the optioanal ElseIf statements are evaluated (there may be zero, one, or more ElseIf statements). If conditional1 is false and an ElseIf statement exists then conditional2

is evaluated. If it is true (see previous paragraph) then Body2 is executed. If it is false, then subsequent ElseIf statements are evaluated.

If all previous If and ElseIf statements are false, the optional Else statements are executed (Body3).

The If statement must be terminated with an End If or Endif statement.

Example

```
Dim a as int
Dim b as int
Input "a:", a
Input "b:", b

If a=b Then
        Print "A is equal to B"
ElseIf a > b Then
        Print "A is Greater Than B"
ElseIf a < b Then
        Print "A is Less Than B"
Else
        Print "ERROR - we should never get here"
End If
```

*Topic : **If / Then***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

If conditional1 Then statement

Description :

The single-line if statement allows an abbreviated conditional check and statement execution. No Else or Elseif is allowed and no End If is necessary.

Example

```
Dim a as int
Dim b as int
Input "a:", a
Input "b:", b

If a=b Then Print "A is equal to B"
If a > b Then Print "A is Greater Than B"
If a < b Then Print "A is Less Than B"
```

*Topic : **Include***

Language Level : CyBasic-3
Effectivity : B2C v2c
Syntax :

Include “filename.b2c”

Description :

Include will insert a file into the source module just as though it were typed in.

Example :

Include “filename.b2c

Topic : Ink

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Ink color

Description :

Ink sets the foreground color of the display. The color parameter can be any numeric expression which evaluates to the numbers 0, 1, 2,3. The Ink will not take effect until either the next Print or Cls command. Use these constants for setting the color

0: WHITE
1: LTGREY
2: DKGREY
3: BLACK

Example :

```
Paper 3
Ink 1
Cls
Print "Top of the screen"
```

Topic ***: Inline***

Language Level : B2C Extension
Effectivity : B2C v2a
Syntax :

Inline *c-code*

Description :

Inline will pass the associated C-code directly through to the output file. This is useful for creating functions in B2C which do not already exist.

Example :

…

Topic ***: Input***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Input {prompt, } variable

Description :

The Input keyword displays an optional prompt and waits for the user to input a value for the variable. If the variable is a character array (string, dim a[32] as char, for example) the entire string of characters will be entered into the array. Otherwise a scalar (non-array) variable must be entered.

Example

```
Dim age as int
Dim name[32] as char
Input "age:", age
Input "name:", name
Print "You are ", name, "and your age is", age
```

Topic : ***Inputxy***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Inputxy x, y, {prompt, } variable

Description :

The Inputxy keyword positions the cursor on the screen at the X, Y coordinates, displays an optional prompt and waits for the user to input a value for the variable. If the variable is a character array (string, dim a[32] as char, for example) the entire string of characters will be entered into the array. Otherwise a scalar (non-array) variable must be entered.

Example

```
Dim age as int
Dim name[32] as char
Inputxy 10, 20 "age:", age
Inputxy 10, 35 "name:", name
Print "You are ", name, "and your age is", age
```

Topic : Int

Language Level : B2C Extension
Effectivity : B2C V3a
Syntax :

Int(s)

Description :

Int converts a string into an integer.

Example :

```
Dim s[10] as char
Dim age as int
Input “enter age”, s
Age = int(s)
```

*Topic : **Inv()***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Inv(n)

Description :

Inv() returns the inverse (reciprocal) of the Double value ‘n’ (in radians).

Example :

```
X = sin(n)
```

*Topic : **Key-function***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

Key(keynumber)

Description :

Key(keynumber) returns TRUE (nonzero) if keynumber is depressed and FALSE (0) otherwise. See Key, below, for a chart of keynumbers. This feature was added in Version 2c to facilitate faster keyboard checking. "Key" is kept for backwards compatibility.

Example :

```
Dim x as int
Dim y as int
Dim z as int
Sprite 1, "lemming.pic"
X=0
Y=0
Z=0
While 1
```

```
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move x,y,z
        Redraw
wend
```

*Topic : **Key-variable***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Key

Description :

Key returns the value of the last depressed key.

Use the #defined values or numbers from the table below
#KEY_A - #KEY_Z
#KEY_0 - #KEY_9
#KEY_ENTER, #KEY_SELECT, #KEY_TAB, #KEY_INS, #KEY_DEL
#KEY_UP, #KEY_DOWN, #KEY_LEFT, #KEY_RIGHT

Key Values	0	1	2	3	4	5	6	7	8	9
30			space							quote
40					comma	dash	period	/	0	1
50	2	3	4	5	6	7	8	9		semi-colon
60		=								
90		(	\	)			back-quote	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z							
260					left-arrow	up-arrow	right-arrow	down-arrow	ins	del
270	tab	select	enter	bspc		shift	fn			

Example :

```
Print "Press any key to continue"
While(key=0)
wend
```

*Topic : **Keyclick***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

Keyclick ON
Keyclick OFF

Description :

Keyclick turns keyclicks on or off

Example :

```
Keyclick Off
```

Topic : Len

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

n = len(string)

Description :

Len accepts a string as a parameter and returns the number of characters in the string.

Example :

```
Dim name[32] as char
Input “Enter your name”, name
Print “Your name has ”, len(name), “ characters in it”
```

Topic : Line

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Line x0, y0, x1, y1

Description :

Line draws a line on the display in the current color (see Ink). The parameters x0 and x1 should be between -80 and 79. The parameters y0 and y1 should be between -50 and 49. Values outside this range will be ineffective.

Example :

```
Paper 3
Ink 1
Cls
'Draw an X
Line -80, -40, 79, 39
Line -80, 39, 79, -40
```

Topic : Load

Language Level : CyBasic-3
Effectivity : B2C v5
Syntax :

Load(filename)

Description :

Load will transfer a file from the .app archive to the Cybiko’s flash memory. This is very useful for ensuring that the file is already on the Cybiko (.dl files in particular). Load will not overwrite a file that already exists in the flash.

Example :

```
Load(“serial.dl”)
```

Topic : Menu

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

Menu var,str1,str2…

Description :

Menu will display a text menu of strings supplied on the command line. The user can select one of the strings from the list using the up and down arrow keys. Pressing the Enter key exits the Menu command. The variable ‘var’ is set to the number (1,2…) of the menu item selected. ‘var’ must be of type int.

The menu uses the currently selected font and the currently selected ink and paper colors to display the menu. The screen is **not** cleared before displaying the menu, so any graphics displayed is used as a background. The menu does, however, draw a blank background just large enough to display the menu, centered in the screen.

Example :

```
Dim selection as int
Top:
Menu selection, “Start”, “Scores”, “Quit”
On selection goto begin, high_score, get_out
Begin:
   Cls
   Print “Starting…”
   …
   goto top
High_score:
   Score
   Goto top
Get_out:
   Exit program
```

*Topic : **Menuxy***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Menuxy x, y, var, str1, str2…

Description :

Menuxy will position (at X, Y) and display a text menu of strings supplied on the command line. The user can select one of the strings from the list using the up and down arrow keys. Pressing the Enter key exits the Menuxy command. The variable ‘var’ is set to the number (1,2…) of the menu item selected. ‘var’ must be of type int.

The menuxy command uses the currently selected font and the currently selected ink and paper colors to display the menu. The screen is **not** cleared before displaying the menu, so any graphics displayed is used as a background. The menu does, however, draw a blank background just large enough to display the menu.

Example :

```
Dim selection as int
Top:
Menuxy 10, 20, selection, “Start”, “Scores”, “Quit”
On selection goto begin, high_score, get_out
Begin:
   Cls
   Print “Starting…”
   …
   goto top
High_score:
   Score
   Goto top
Get_out:
   Exit program
```

Topic : Mid

Language Level : CyBasic-2
Effectivity : B2C v2

Syntax :

Mid destination, source, start {, length}

Description :

Mid will copy into the destination string from the source string starting at the start position. If the length parameter is not specified then the entire length of the source string is copied. If length is specified then only length characters are copied.

Example :

```
Dim name[32] as char
Dim first5[10] as char
Input "Name", name
Mid first5, name, 0, 5
Print "First part=", first5
```

Topic : Move

Language Level : B2C Extension

Effectivity : B2C v2c

Syntax :

Move n, x, y [, z [,mode]]

Description :

Move moves a sprite on the screen. 'n' is the sprite number (0-32), 'x' is the horizontal coordinate, 'y' is the vertical coordinate, 'z' is optional and is the bitmap within the sprite to display (this is usually used for animation), finally the optional 'mode' is the direction to 'flip' the bitmap before displaying (FLIP_X for horizontal flip, FLIP_Y for vertical, and FLIP_X+FLIP_Y for both). This feature allows you to reuse the bitmap. For example, when walking to the left, a character should be flipped horizontally while not flipped at all while walking right.

You must do a redraw to see the results of the move command. Because of the cost of doing a redraw, it is advisable to do all the moves for a 'turn' and then a single redraw.

Example

```
Dim x as int
Dim y as int
Dim z as int
Dim mode as int
Sprite 1, "lemming.pic"
X=0
Y=0
Z=0
Mode=0
While 1
        If key(#KEY_LEFT) then
                x=x-1
                mode = FLIP_X
        endif
        If key(#KEY_RIGHT) then
                x=x+1
                mode=FLIP_Y
        Endif
        Move 1,x,y,z,mode
        Redraw
wend
```

*Topic : **Music***

Language Level : B2C Extension

Effectivity : B2C v2c

Syntax :

Music [Background | Foreground], "filename.mus"
Music [Background | Foreground], [STOP | PLAY]

Description :

The Music command loads, plays, and stops music playing on the Cybiko. There are 2 "channels" of sound play on the Cybiko - the Background and the Foreground. The first form of the command loads a file which was bound into the app using the filer.exe command. The second form of the command starts or stops music playing.

Example

```
Dim x as int
Dim y as int
Dim z as int
Dim a as int
Dim b as int
Music Background, "lemmings.mus"
Music Background, Play
Sprite 1, "lemming.pic"
Sprite 2, "other.pic
X=0
Y=0
Z=0
A=-80
B=-50
Move 2, a, b
While 1
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move x,y,z
        Beep 0
        Vibrate 0
        If Collision(1,2) then
                beep 1
                vibrate 128
        Redraw
wend
```

Topic : ***Nextfile***

Language Level : B2C Extension
Effectivity : B2C v3e
Syntax :

rc = Nextfile *(filename)*

Description :

The Nextfile function will retrieve the next file established by the Findfile command. It returns FALSE when there are no more files to find and TRUE otherwise.

Example :

```
Dim s[32] as char
Findfile “*.app”
While (nextfile(s))
        Print s
wend
```

Topic : ***On-Gosub***

Language Level : B2C Extension
Effectivity : B2C v3

Syntax :

On n Gosub sub1, sub2…

Description :

On Gosub will call a subroutine based on the value of 'n'. If n=1 then sub1 is called, if n=2 then sub2 is called, etc… If n=0 or greater than the number of supplied subs, processing continues at the next command.

Example :

```
Sub start_sub
        Print "started"
End sub
Sub scores_sub
        Score
End sub
Sub quit_sub
        Exit Program
End Sub
Dim n as int
Menu n, "Start", "Scores", "Quit"
On n Gosub start_sub, scores_sub, quit_sub
```

*Topic : **On-Goto***

Language Level : B2C Extension

Effectivity : B2C v3

Syntax :

On n Goto label1, label2, …

Description :

On Goto will branch to a label based on the value of 'n'. If n=1, control will be passed to label1, if n=2 control will be passed to label2, etc…

Example :

```
Dim selection as int
Top:
Menu selection, "Start", "Scores", "Quit"
On selection goto begin, high_score, get_out
Begin:
   Cls
   Print "Starting…"
   …
   goto top
High_score:
   Score
   Goto top
Get_out:
   Exit program
```

*Topic : **OnMessage***

Language Level : B2C Extension

Effectivity : B2C v3d

Syntax :

sub onMessage(cyid as long, msgno as int, buf[] as char)

Description :

OnMessage is a user-supplied function. It is called "spontaneously" (like an interrupt) whenever a message enters the message queue for the currently running app. (In fact the message queue is checked during calls to keyboard input routines). Do not call keyboard functions (key, getchar, input, etc... during onmessage() calls)

Example :

```
sub onMessage(cyid as long, msgno as int, buf[] as char)
        if buf[0] = '1' then beep 10
        else beep -1
end sub
```

See Also:

SendMessage, UserMenu, toCyid, toUser, Tokenize

Topic : Open

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Open pathname {For mode} As filenumber

Description :

Open opens a file for reading, writing or append access. Pathname is a string expression (constant quoted string or character array variable) which points to the pathname of a file. Optional mode is one of READ, ARCHIVE, WRITE, or APPEND. Filename is an integer expression which indicates the handle for the file. Use filenumber in future calls to get, put, and close.

READ=open a file for read access only
ARCHIVE=open a file from the archive for read access only
WRITE=open a file for write access only
APPEND=open a file for writing to the end of the file

Errors:

#ERROR_FILE_NUMBER – the file number is outside the valid range
#ERROR_FILE_ALREADY_OPEN – the file is already open and should first be closed
#ERROR_FILE_OPEN – there was a problem opening the file for READ or APPEND access
#ERROR_FILE_CREATE – there was a problem opening the file for WRITE access

Example :

```
Dim foo as int
Foo = 1
Open "a.dat" For Write as 1
if error then exit program
Put 1, 0, foo
Close 1
```

*Topic : **OPTION-3DROOMS***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

OPTION 3DROOMS n

Description :

OPTION 3DROOMS will set the maximum number of rooms available for the 3D software. The default is 8 rooms.

Example :

```
OPTION 3DROOMS 16
```

*Topic : **OPTION-3DSPRITES***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

OPTION 3DSPRITES n

Description :

OPTION 3DSPRITES sets the maximum number of sprites available for the 3D software. The default is 8 sprites.

Example :

OPTION 3DSPRITES 16

*Topic : **OPTION-C_STRINGS***

OPTION CYBASIC_STRINGS (default)

Language Level : B2C Extension

Effectivity : B2C v2c

Syntax :

OPTION C_STRINGS
OPTION CYBASIC_STRINGS

Description :

The C compiler offers a powerful string formatting capability. Characters placed in a literal string can be preceded by the backslash character (\) also known as the escape character. Using this escape character special characters (like \n for newline and \003 for ^C) could be inserted into the middle of a string.

However Cybasic does not offer this capability. To solve this delemma, the OPTION C_STRINGS command was introduced. If OPTION C_STRINGS appears at the top of a program, the escape character is interpretted as a C string. However if OPTION CYBASIC_STRINGS appears at the top of a program, the escape character has no special meaning. The default is OPTION CYBASIC_STRINGS.

Example :

```
OPTION C_STRINGS
Print "\"Hello World\""
```

Results:

"Hello World"

*Topic : **OPTION-C_COORDS***

OPTION CYBASIC_COORDS (default)

Language Level : B2C Extension

Effectivity : B2C v3

Syntax :

OPTION C_COORDS
OPTION CYBASIC_COORDS

Description :

Normally, the B2C screen coordinates emulate the Cybasic default – (-80,-50) (lower left) to (79,49) (upper right). This can be selected with the compiler option OPTION CYBASIC_COORDS (default). The C coordinate system is (0,0) (upper left) to (159,99) (lower right). This mode can be selected with the compiler option OPTION C_COORDS.

Example :

```
OPTION C_COORDS
Line 0,0,199,99  'draw an x
Line 199,0,0,99
```

*Topic : **OPTION-ESCAPE***

Language Level : B2C Extension

Effectivity : B2C v3a

Syntax :

OPTION ESCAPE ON (default)
OPTION ESCAPE OFF

Description :

The OPTION ESCAPE function enables and disables the escape key checking that occurs inside a for or while loop. Escape key checking can interfere with normal key() function calls by swalling key events. For interactive games OPTION ESCAPE OFF is recommended.

Example :

OPTION ESCAPE OFF

*Topic : **OPTION-HELP***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

OPTION HELP ON (default)
OPTION HELP OFF

Description :

OPTION HELP controls the action of the HELP key. With OPTION HELP ON, when the user presses the HELP (?) key the help file is displayed. With OPTION HELP OFF the HELP key is ignored.

Example :

OPTION HELP OFF

*Topic : **OPTION-MAIN***

Language Level : B2C Extension
Effectivity : B2C v4
Syntax :

OPTION MAIN ON (default)
OPTION MAIN OFF

Description :

The OPTION MAIN function enables and disables main function processing. Normally a B2C file uses all commands not in functions as the main program. Turning OPTION MAIN OFF will ignore all such commands. This is used to create modular B2C files with just functions in them. Heretofore B2C recognized only 1 source module. Now it is possible to have a library of functions that are reusable.

Modules with OPTION MAIN OFF inherits the other options from the MAIN module.

NOTE: No provision for the prototyping of functions has been made. This will come in a later release. The Cybiko compiler will generate a abnormal app without proper prototypes. Until a provision is made for them, use the C prototypes and “outline” them.

Example :

OPTION MAIN OFF

*Topic : **OPTION-MULTITASK***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

OPTION MULTITASK ON (default)
OPTION MULTITASK OFF

Description :

OPTION MULTITASK OFF is mainly for the HELP key. When the HELP key is pressed it spawns a new thread which displays the help info. Normally the video game or app continues to run in MULTITASKING MODE. This is bad because it can erase the help screen and / or cause

the player to lose "lives" or "health points" etc... because the game is continuing on and the player cannot see it.

Specifying OPTION MULTITASK OFF means that when the HELP screen is displayed, the game pauses. Now this may not be desired. For example a CHAT application could drop messages. So, MULTITASKING is a compile-time option.

Example :

```
OPTION MULTITASK OFF
```

*Topic : **OPTION-SHOW***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

```
OPTION SHOW OFF
OPTION SHOW ON (default)
```

Description :

Normally the B2C graphics commands (line, point, printxy) immediately display their results on the Cybiko screen. This mode is OPTION SHOW ON and is the default. However, if OPTION SHOW OFF is selected then the graphics commands are buffered until a Redraw Show command is executed. This is a significant performance enhancement.

Example :

```
OPTION SHOW OFF
For I=0 to 99
        Line 0,0,79,i  'draw a fan
Next
Redraw Show
```

*Topic : **OPTION-SPRITES***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

```
OPTION SPRITES n
```

Description :

The OPTION SPRITES command determines the maximum number of sprites that can be processed by B2C. The default is 32.

Example :

```
OPTION SPRITES 16
For I=0 to 15
        Sprite I, "sprite.pic"
Next
```

*Topic : **Outline***

Language Level : B2C Extension
Effectivity : B2C v3a
Syntax :

Outline *c-code*

Description :

Outline will pass the associated C-code directly through to the output file. This is useful for creating functions in B2C which do not already exist. Outline code is placed in the .c file *outside* the main function.

Example :

…

Topic : Page

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Page 0/1

Description :

Page will set the current graphics page. The Cybiko has 2 graphics pages (page 0 and page 1). The default is Page 0. All graphics commands will be displayed on the currently selected page. Changing the graphics page starts rendering on that page. You can render graphics on one page while displaying the other one (this requires OPTION SHOW OFF and REDRAW SHOW). This is called "double-buffering".

Example :

```
OPTION SHOW OFF
page 0   'display page 0
cls ' clear page 0
rect 0,0,40,40 'draw a rectangle on page 0
redraw show ' show page 0

page 1 'select page 1
cls ' clearing page 1 does not affect page 0
rectfill -40,40,40,40 ' draw a filled rect on page 1

'now alternate between the two pages
while (key(#KEY_ENTER)=0)
       page 0 ' select page 0
       redraw show ' show page 0
       wait 10
       page 1 'select page 1
       redraw show 'show page 1
       wait 10
wend
```

*Topic : **Pagecopy***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Pagecopy from, to, x, y, w, h

Description :

The Cybiko has 2 graphics pages (see the Page command, above). Pagecopy will transfer a rectangle from one page to another. The graphics "patch" will be transferred from the FROM page to the TO page and to the same X, Y coordinates. This is useful for quickly rendering a piece of a background page to the currently displayed page, especially video games. One can erase a sprite from page 0 by copying a rectangle from the background screen in page 1. This is faster than repainting the entire background sprite and repainting all the character sprites.

Example :

```
OPTION SHOW OFF
OPTION ESCAPE OFF
OPTION C_COORDS

dim x as int
dim x1 as int
dim y as int

Sub redraw_mario
    pagecopy 1, 0, x1, y, 20, 20 'erase mario from old x,y with the background
    move 1,x,y 'paint mario
        redraw 1
    x1 = x 'set the old x to the current x
    redraw show ' show page 0
end sub

Sprite 0, "background.pic"
Sprite 1, "mario.pic"
```

```
page 1 'switch to page1
move 0, 0, 0 ' paint the background to page 1
redraw 0

page 0 ' back to page 0
move 0, 0, 0
redraw 0 ' paint the background to page 0

' initialize mario's position
x=0
x1=x
y=50

move 1, x, y  'paint mario
redraw 1
redraw show    'show page 0 to the user

while true
        if key(#KEY_LEFT) then
                x=x-1
                redraw_mario
        endif
        if key(#KEY_RIGHT) then
                x=x+1
                redraw_mario
        endif
    if key(#KEY_SPACE) then exit program
wend
```

*Topic : **Paper***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Paper color

Description :

Paper sets the background color of the display. The color parameter can be any numeric expression which evaluates to the numbers 0, 1, 2,3. The Paper will not take effect until either the next Print or Cls command.

0: WHITE
1: LTGREY
2: DKGREY
3: BLACK

Example :

Paper 3
Ink 1
Cls
Print "Top of the screen"

*Topic : **Point***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Point x, y

Description :

Point sets a single pixel on the display in the current color (see Ink). The parameter x should be between -80 and 79. The parameter y should be between -50 and 49. Values outside this range will be ineffective.

Example :

Paper 3
Ink 1
Cls

Point 0, 0 ' set a dot in the center of the display

*Topic : **Print***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Print {expr {,expr…}}

Description :

The Print keyword displays a line of output on the Cybiko display. Print will display any expression. If the text of the display is greater than the number of lines in the Display, then the display will scroll up to reveal more text. The number of lines in the display is dependent on the Font used (see Font command). The number of characters per line is also dependent on the Font used. If the number of characters printed is greater than the width of the display, the output will be truncated. Print automatically adds a newline to the end of the line.

Example

```
Dim age as int
Dim name[32] as char
Input "age:", age
Input "name:", name
Print "You are ", name, "and your age is", age
```

*Topic : **Printno***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Print# n {expr {,expr…}}

Description :

The Print# keyword writes a line of output on file number ‘n’. Print# will write any expression. Print# automatically adds a newline to the end of the line. This is similar to the Print command for the display but for files.

Error :

#ERROR_FILE_NUMBER – the file number specified is out of the valid range
#ERROR_FILE_UNOPENED – the file has not been opened
#ERROR_FILE_WRITE – there was an error while attempting to write the file

Example :

```
Dim age as int
Dim name[32] as char
Input "age:", age
Input "name:", name
Print# 1, "You are ", name, "and your age is", age
if error then exit program
```

*Topic : **Printxy***

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Printxy x, y {, expr …}

Description :

Printxy will print the expressions (see Print) at the x and y coordinates specified.

Example :

```
Printxy 0, 0, "Center"
```

Topic : Put

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

Put filenumber, {bytepos},{ variable}

Description :

Put writes a variable to a file at the current or specified byte position. The filenumber must be the number of a file already openned for Write or Append access. If bytepos is specified, the variable will be written at that position in the file. If bytepos is unspecified, then the current position is used. If the variable is left unspecied, then the file pointer is changed to bytepos and no data is written.

Error :

#ERROR_FILE_NUMBER – the file number is outside the valid range
#ERROR_FILE_UNOPENED – the file has not been opened
#ERROR_FILE_WRITE – the file could not be written
#ERROR_FILE_SEEK – the file could not be positioned

Example :

```
Dim foo as int
Foo = 1
Open "a.dat" For Write as 1
Put 1, 0, foo
if error then exit program
Close 1
```

*Topic : **Redraw***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

Redraw All
Redraw n
Redraw Show

Description :

The Redraw command has 3 modes of operations

Redraw All :

Redraw All will redraw all sprites from Sprite 0 through Sprite n (which is defined by the Option Sprites command). Sprite 0 is drawn first. If it is not defined then the entire screen is erased with the currently defined paper color. If Sprite 0 is defined then it is tiled against the background. Setting the X,Y coordinates of Sprite 0 (with the Move Command) can create a scrolling effect. The result of the Redraw command is immediately visible on the display.

Redraw n:

Redraw n will draw only the sprite ‘n’ onto the display. However it will not immediately show the result of the redrawing. A call to Redraw Show is needed to make the result visible. This is a performance enhancement. You should redraw all the sprites in the current frame using Redraw n and call Redraw Show only after all sprites have been Redrawn.

Redraw Show:

For all graphics commands (line, point, printxy), the user will see the result of the command only if OPTION SHOW ON is set. If OPTION SHOW OFF is set then Redraw Show will update the display with the results of the graphics command. This is also true of sprites drawn with the Redraw n command.

Example

```
Dim x as int
Dim y as int
```

```
Dim z as int
Sprite 1, "lemming.pic"
X=0
Y=0
Z=0
While 1
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move 1,x,y,z
        Redraw
Wend
```

*Topic : **Remove***

Language Level : Cybasic3
Effectivity : B2C v5
Syntax :

Remove filename

Description :

Remove will delete the named file from the Cybiko's flash memory.

Example :

Remove "foo.dat"

*Topic : **Rename***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

Rename old-filename, new-filename

Description :

Rename will rename the old filename to the new filename.

Example :

Rename "old.dat", "new.dat"

*Topic : **Score***

Language Level : B2C Extension
Effectivity : B2C v3
Syntax :

Score
Score high_score

Description :

The Score command has two modes. If no parameter is supplied, it will show the top 5 scores for the game in the current font and the current ink and paper colors. If a high_score is supplied, then the high_score is compared to existing high scores in the score.inf file and will replace one of them with the new high_score (if the high_score is higher).

Note: You must include a score.inf file (supplied) in your .list file.

Example :

```
Dim high_score as long
…
High_score = High_score + 100
…
Score high_score ‘set the high score
cls
Score ‘show the high scores
```

```
While key(#KEY_ENTER) = 0
wend
```

Topic : Rect

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

```
Rect X, Y, W, H
```

Description :

Rect will draw a rectangle at location X, Y with a width of W and a height of H.

Example :

```
Rect 0,0,40,40
```

*Topic : **Rectfill***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

```
Rectfill X, Y, W, H
```

Description :

Rect will draw a filled rectangle at location X, Y with a width of W and a height of H.

Example :

```
Rectfill 0,0,40,40
```

*Topic : **Right***

Language Level : CyBasic-2
Effectivity : B2C v2
Syntax :

```
Right destination, source, length
```

Description :

Right will copy into the destination string from the source string for length characters.

Example :

```
Dim name[32] as char
Dim last5[10] as char
Input "Name", name
Right last5, name, 5
Print "Last part=", last5
```

Topic : Rnd

Language Level : CyBasic-2
Effectivity : B2C v2
Syntax :

```
Rnd(x)
```

Description :

Rnd() returns a random number from 0 to x.

Example :

```
Print rnd(100) 'print a random number from 0-100
```

*Topic : **Sendmessage***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

```
Sendmessage cyid, message_number, appname, text_string
```

Description :

Sendmessage sends an RF messages as a “text_string” to the Cybiko with associated “cyid”. The awaiting Cybiko must be running the app called “appname”. The message will appear in the receiving Cybiko’s OnMessage function. CyIDs can be selected using the UserMenu function. CyIDs must be a Long integer. Message numbers are in the range 0-1023.

THE RECEIVING CYBIKO MUST BE RUNNING APPNAME OR THE MESSAGE WILL NOT GET THROUGH.

Example :

Sendmessage cyid, 0, “chat”, “hello world”

See Also:

OnMessage, UserMenu, toCyid, toUser, Tokenize

*Topic : **Sin()***

Language Level : CyBasic-1
Effectivity : B2C v5
Syntax :

Sin(n)

Description :

Sin() returns the sine of the Double value ‘n’ (in radians).

Example :

X = sin(n)

*Topic : **Sprint***

Language Level : CyBasic-1
Effectivity : B2C v3d
Syntax :

Sprint varname{expr {,expr…}}

Description :

Sprint does the same operations as Print but the result goes into the Variable “varname”

Example

```
Dim age as int
Dim name[32] as char
Dim result[64] as char
Input "age:", age
Input "name:", name
Sprint result, "You are ", name, "and your age is", age
```

*Topic : **Sprite***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

Sprite n, "filename"
Sprite n

Description :

A Sprite is an animated object on the Cybiko screen. The first form of the Sprite command loads a ".pic" file into memory. The value 'n' indicates which sprite (0-32) is being loaded. This file should have been created using the SDK program "2pic". Once in memory, the Move command can be used to move and animate the Sprite.

The second form deletes the sprite from memory.

Example

```
Dim x as int
```

```
Dim y as int
Dim z as int
Sprite 1, "lemming.pic"
X=0
Y=0
Z=0
While 1
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move x,y,z
        Redraw
wend
```

*Topic : **Sqrt()***

Language Level : CyBasic-1
Effectivity : B2C v5
Syntax :

```
Sqrt(n)
```

Description :

Sqrt() returns the square root of the Double value ‘n’.

Example :

```
X = sqrt(n)
```

*Topic : **Stringheight()***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

```
n = stringheight(string)
```

Description :

Stringheight returns the number of y-pixels the ‘string’ will take if displayed in the current font.

Example :

```
OPTION C_COORDS
Dim x as int
Dim y as int
Dim s[32] as char
Input “enter your name”, s
x = stringwidth(s)
y = stringheight(s)
‘ center text on the screen
cls
printxy (160-x)/2, (100-y), s
```

*Topic : **Stringwidth()***

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

```
n = stringwidth(string)
```

Description :

Stringwidth returns the number of x-pixels the ‘string’ will take if displayed in the current font.

Example :

```
OPTION C_COORDS
Dim x as int
Dim y as int
```

```
Dim s[32] as char
Input “enter your name”, s
x = stringwidth(s)
y = stringheight(s)
‘ center text on the screen
cls
printxy (160-x)/2, (100-y), s
```

Topic : Sub

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

```
Sub subname {(param1{[]} {As datatype1} {, param2 {As datatype2} …})}
        {Exit Sub}
        Body
End Sub
```

Description :

A Sub is very much like a Function, but it returns no value and therefore has no associated datatype

Sub initiates the definition of a new subroutine . subname is the name of the subroutine. An optional parameter list may be specified. Zero, one, or more parameters may be specified. Each parameter may have a datatype as specified in Dim. However, no array parameters may be passed. If the datatype is not specified, the variable is defined as Double

If the Exit Sub statement is encountered, the function will return immediately.

With version 3a of B2C, single-dimensioned arrays may be passed into subs. The format of the parameter is p[] – indicating an array of arbitrary length. This now allows strings to be passed into functions and subroutines. See the example under ‘Function’

Example

```
Sub menu
        Print "1) Attach"
        Print "2) Retreat"
        Print "3) Plead for Life"
        Print "0) Quit"
End Sub

Dim x
Call Menu
Input x
```

*Topic : **Tan()***

Language Level : CyBasic-1
Effectivity : B2C v5
Syntax :

```
Tan(n)
```

Description :

Tan() returns the tangent of the Double value ‘n’ (in radians).

Example :

```
X = tan(n)
```

*Topic : **ToCyid***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :
Cyid = ToCyid(username)
Description :
ToCyid will convert a username (nickname) into a numeric CyID. CyId must be a long integer
Example :
cyid = ToCyid(username)
See Also:
SendMessage, OnMessage, UserMenu, toUser, Tokenize

*Topic : **Tokenize***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :
Tokenize string, separators, var1, var2, var3...
Description :
Tokenize will split a "string" into smaller strings based on the "separators". This is useful for breaking up comma-separated (or other) strings into individual variables.
Example :
Tokenize "1|2|3|4", "|", var1, var2, var3, var4
See Also:
SendMessage, OnMessage, UserMenu, toUser, toCyid

*Topic : **ToUser***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :
name = ToUser(cyid)
Description :
ToUser will convert a numeric CyID into a username (nickname). CyID must be a Long Integer.
Example :
cyid = ToCyid(username)
See Also:
SendMessage, OnMessage, UserMenu, toCyid, Tokenize

Topic : True

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :
true
Description :
Constant value returns true – or 1.
Example :
While(true)
‘ something cool
wend
NOTE: This is recommended over while(1)

Topic : Type

Language Level : Cybasic-2

Effectivity : B2C v3a
Syntax :

```
Type name
  Member{[a{,b}]} as type
  …
End Type
```

Description :

Type allows the creation of user-defined data types. Each line after the Type command is a member definition, much like a Dim command. Once a Type is defined variables can be defined as that type. Members are dereferenced with dot specifiers.

Example :

```
Type coord
  X as int
  Y as int
End type

Type line
  P0 as coord
  P1 as coord
End Type

Dim line0 as line

Line0.p0.x=0
Line0.p0.y=0
Line0.p1.x=50
Line0.p1.y=50
```

Topic : ***UserMenu***

Language Level : B2C Extension
Effectivity : B2C v3d
Syntax :

```
Cyid=UserMenu(prompt, exception_list)
```

Description :

UserMenu will display a list of nearby Cybiko units for the user to select. The "prompt" string is displayed in the user menu and all nearby Cybiko's – except the Cybiko's in the "exception_list" – will be displayed for selection. The exception list is an array of Long integers. All CyIDs in the array will be excluded from the menu. The last CyID in the list must be zero (a Null Terminator). Placing a Zero in place of the exception list will display all Cybiko's nearby. If the user exits the menu by pressing ESC, then a Zero is returned.

Example :

```
cyid = usermenu("Select Chat Client", 0)
```

See Also:

SendMessage, OnMessage, toCyid, toUser, Tokenize

Topic : ***Vibrate***

Language Level : B2C Extension
Effectivity : B2C v2c
Syntax :

```
Vibrate n
```

Description :

Vibrate causes the Cybiko to vibrate. 'n' is the strength of the vibration and is from 0 to 255. A vibrate of 0 will cause the vibrations to stop. A vibrate of 128 is medium and a vibrate of 255 is maximum.

Example :

```
Dim x as int
Dim y as int
Dim z as int
Dim a as int
Dim b as int
Sprite 1, "lemming.pic"
Sprite 2, "other.pic
X=0
Y=0
Z=0
A=-80
B=-50
Move 2, a, b
While 1
        If key(264) then x=x-1
        If key(266) then x=x+1
        Move x,y,z
        Beep 0
        Vibrate 0
        If Collision(1,2) then
                beep 1
                vibrate 128
        Redraw
wend
```

Topic : Wait

Language Level : CyBasic-1
Effectivity : B2C v2
Syntax :

```
Wait n
```

Description :

Wait causes a delay in a B2C program for n/10 seconds.

Example :

```
Print "Hello"
Wait 10 ' one second delay
Print "World"
```

*Topic : **While***

Language Level : CyBasic-2
Effectivity : B2C v2
Syntax :

```
While(conditional)
        Body
        Exit While
        Body
Wend
```

Description :

While initiates a loop which continues while the conditional (see IF) is true. Once the conditional is false, the loop terminates.

You may terminate the loop early by executing an Exit While command.

Example :

```
While(key = 0)
```

```
        Print "… Waiting!"
        Wait 3
        If (Key(#KEY_ENTER)) Then Exit While
    Wend
```

Topic : Wrap

Language Level : B2C Extension
Effectivity : B2C v5
Syntax :

```
n = wrap(x,y,w,h,align,wrap,string)
```

Description :

Wrap will display the ‘string’ at the x/y coordinates inside a rectangle with width ‘w’ and height ‘h’. The text will fill the rectangle with the text and will perform either character wrapping or word wrapping. Also, each line of the text will be aligned either on the left, centered, or on the right.

‘align’ can be one of LEFT, CENTER, or RIGHT
‘wrap’ can be one of CHAR or WORD

Wrap returns the number of characters actually printed in the rectangular region.

Example :

```
OPTION C_COORDS
‘ display a centered title
dim n as int
n = wrap(0,0,160,15,CENTER,CHAR,”Title Text”)

‘display a block of text left justified and word-wrapped
rect(0,15,80,30)
n = wrap(0,15,80,30,LEFT, WORD, “This is a test of the emergency broadcast system.”)
```

Chapter 31: B2C RESERVED WORDS

A list of words used by the
B2C Language
Which cannot be used by applications

3dcollision	3dget	3dmove	3dredraw	3drmove	3droom	3dsprite	3dwall
abs	acos	acosh	actan	actanh	all	and	append
as	asin	asinh	atan	atanh			
background	beep	black	by_ref	by_val			
c_coords	c_strings	call	camera	char	close	cls	
collision	cos	cosh	cybasic_coords		cybasic_strings		
dialog	dim	dkgrey	double				
else	elseif	end	endif	error	escape	exit	exp
false	filelist	filename	findfile	flip_x	flip_y		
font	for	foreground	function				
get	getchar	getkey	gosub	goto			
if	ink	inline	input	int	int		
key	keyclick						
line	log	log10	log2	long	ltgrey		
main	menu	mid	mod	move	music		
neq	next	nextfile	not				
off	on	open	option	or	outline		
paper	play	point					
pow	pow2	print	printxy	program	put		
read	redraw	right	rnd				
score	sendmessage	show	sin	sinh	sprint		
sprite	sprite_get	sprites	sqr	sqrt	step	stop	sub
tan	then	to	tocyid	tokenize	touser	true	type
usermenu							
vibrate							
wait	wend	while	white	write			

Chapter 32: C RESERVED WORDS

A list of words used by the
C Programming Language
Which cannot be used by applications

and	and_eq	asm	auto				
bitand	bitor	bool	break				
case	catch	char	class	compl	const	const_cast	continue
default	delete	do	double	dynamic_cast			
else	enum	explicit	export	extern			
false	float	for	friend				
goto							
if	inline	int					
long							
main	mutable						
namespace	new	not	not_eq				
operator	or	or_eq					
private	protected	public					
register	reinterpret_cast		return				
short	signed	sizeof	static	static_cast	struct	switch	
template	this	throw	true	try	typedef	typeid	typename
union	unsigned	using					
virtual	void	volatile					
wchar_t	while						
xor	xor_eq						