

# Juiced.GS

A quarterly Apple II journal

September 2015

Concentrate

## GSoft BASIC



*Originally released in 1998, GSoft BASIC for the Apple IIgs is a powerful evolution of Applesoft BASIC, while remaining a fun and accessible programming language. Learn more in Eric Shepherd's six-part series.*

**Inside**  
this issue ...

**On track:**  
Learning the  
language  
Pages 7-14

**Toolbox:**  
Mastering  
the IIgs  
Pages 34-39

**And much,  
much more!!**



## My Home Page

# Reinvention

Welcome to *Juiced.GS Concentrate: Programming with GSoft BASIC*! I'm pleased to present this voluminous compilation, written entirely by renowned Apple II programmer Eric Shepherd. Everything you see from page 4 on is his handiwork.

This six-part series originally debuted in the December 1998 issue of *Juiced.GS*, which included a 3.5" floppy disk with a demo version of GSoft BASIC, courtesy Mike Westerfield at The Byte Works. That software is included with this PDF with permission of The Byte Works. But if you received this PDF for free with the Opus II software collection, then you also have the full version of GSoft; these tutorials will work perfectly in either edition.

While the content of this series has not been edited since its original publication, we've cleaned up the layout a bit. However, formatting code can be challenging, with line breaks occasionally causing confusion.

Where possible, type sizes have been adjusted to allow entire lines of code to actually appear on the desired line. Despite these adjustments, some lines of GSoft BASIC code had to be broken into two lines. To make sure you are seeing, understanding, and typing your code into the GSoft editor properly, keep these simple guidelines in mind:

- All new lines of code are indented.
- The second line of any strings of code that are broken will begin at the far left side of the column and won't be indented. Also, assume there is a space (spacebar) in that line of code where the break occurred, and type that space accordingly when entering the code into GSoft.

- Whereas the narrative of this article is presented in New Century Schoolbook typeface, the code is presented in Courier, to help make it stand out clearly and distinctly.

Further, this issue's content comes from the original text, not from an OCR'ed scan. If you previously bought PDFs of *Juiced.GS* Volumes 1–6, you'll find the text in this *Concentrate* to be of a higher fidelity, suitable for copying and pasting into your favorite emulator or text editor. My thanks to Max Jones, Paul Zaleski, and Ewen Wannop for providing this text, and to Giselle Marks for taking the cover photo of her son, Ian.



Although I don't identify as a programmer, GSoft BASIC will always have a special place in my heart. It debuted at my first KansasFest back in 1998, and I immediately adopted it for the inaugural HackFest competition. I'd just completed my first year of college studies in computer science, and I was eager to apply those lessons to my favorite computer. In the course of my hacking, I even found a bug in GSoft BASIC! I'll always remember Mike Westerfield fixing the bug and handing me the newly released v1.0.1. Talk about customer support!

My attempt at writing a GSoft BASIC version of the classic party game Boggled left me, ah, boggled! But I placed third, with the judges acknowledging my audacity in entering the competition with a language that was, at the time, only days old.

I hope you enjoy GSoft BASIC as much as I did.

**Ken Gagne**  
*Editor-in-Chief*



# Retrospective: GSoft BASIC

## Plotting the evolution of GSoft

By Mike Westerfield

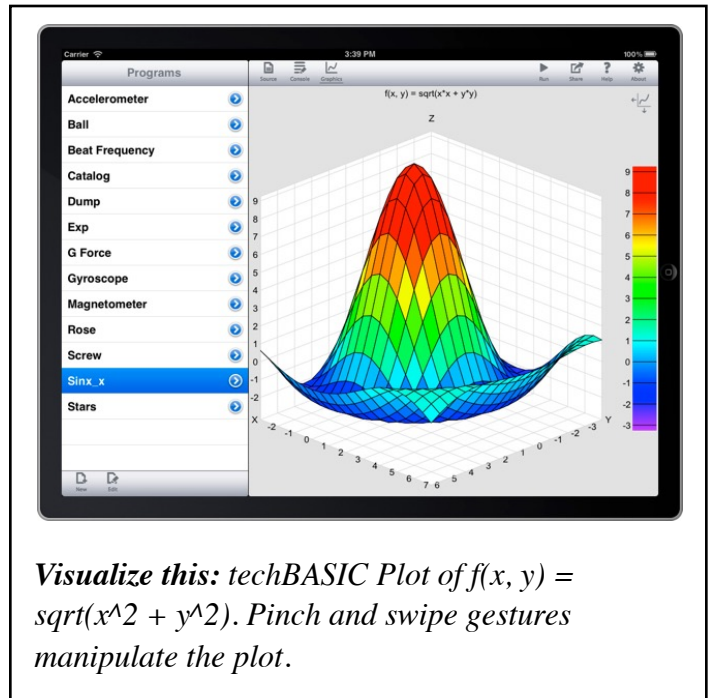
Like a lot of retro computing hardware and software, GSoft BASIC is generating new interest. I thought it would be fun to fill you in on how GSoft BASIC has evolved since the initial Apple IIGS version—as the direct descendants of GSoft BASIC are still in use!

The Apple IIGS version of GSoft was released in 1998, just as the Apple IIGS was fading away. It's an interpreter written in 65816 assembly language, with a few C support programs. GSoft BASIC gives you both an Applesoft-like environment and an ORCA environment for editing and running programs, including toolbox programs.

Some of you probably remember HyperStudio and might be aware that it vanished for several years after Havas bought out Roger Wagner. During that time, I got together with Roger's brother and some of his friends. They had started a company called Tech4Learning, which still produces great programs for lower grade-level education. I wrote a multimedia authoring tool for them called MediaBlender. One of the things MediaBlender needed was a scripting language. After some discussion, we decided to use BASIC. I translated GSoft BASIC to Java and released it as the scripting language for MediaBlender. It was called Boost and was released circa 2005.

Around that time, I also started working for ARES Corporation, an engineering consulting firm that was developing a program called AVERT. I signed on right after the proof-of-concept phase to help turn the concept into a viable program. AVERT is a high-end simulation that uses Monte Carlo, artificial intelligence, and agent-based modeling to simulate attacks on facilities. It is usually used to evaluate different security postures or facility upgrades to see if they deliver the desired probability that the defenders will succeed in protecting the site. It has been used to evaluate security at some of the most sensitive facilities in the world. It too needed a scripting language. ARES licensed Boost in 2008 and used the scripting language for AVERT. As far as I am aware, they still do.

Times change, and so did GSoft BASIC. When I left ARES, I started working on a new translation of GSoft



*Visualize this: techBASIC Plot of  $f(x, y) = \sqrt{x^2 + y^2}$ . Pinch and swipe gestures manipulate the plot.*

BASIC, this time for the iPhone and iPad. I moved the Java version to C. One of the things that a lot of people have forgotten about BASIC is that it was originally designed as an easier version of FORTRAN, intended for use by scientists and engineers for numeric processing. It had native support for matrices and vectors, assignment of arrays, and so forth. All this was necessarily lost when Bill Gates squeezed the language into the small ROM footprint of early microcomputers. I put all of that back in, along with high-end interactive graphics support and easy connectivity to the inbuilt sensors in the iPhone, as well as support for Bluetooth LE and Wi-Fi connectivity. techBASIC, first released in 2011 and updated again just this month (August 2015), is a wonderful tool for developing technical applications on the iPhone and iPad. You can find it on the App Store or at

<http://www.byteworks.us/>

And it all started with GSoft BASIC!

***Mike Westerfield founded The Byte Works in 1981 while in the United States Air Force, which he left at the rank of captain to focus on The Byte Works full-time.***

# Getting Back to BASIC

## *GS programmers speak a new language*

Although BASIC has largely fallen by the wayside in recent years—criticized as being "too slow" or "too unstructured"—there are some areas in which BASIC is still very popular. For example, it's still an excellent teaching tool for beginning programmers, and it has become popular as a scripting language. Microsoft Visual Basic under Windows, and REAL Software's REALbasic on the Macintosh, are popular development environments for those platforms.

BASIC on the Apple IIGS has never been a real solution for creating serious software. Applesoft BASIC has remained woefully unchanged since the dawn of the Apple II era, and lacks any support for using the enhanced features of the Apple IIGS, such as sound, super-hires graphics, or extra memory. There have been other BASIC implementations (such as TML Basic, from the early days, or Micol Advanced BASIC), but neither of these really handles the Apple IIGS Toolbox with any degree of grace.

GSoft BASIC, released this summer at KansasFest 1998, is the Byte Works' new BASIC interpreter for the Apple IIGS. It's important to note that GSoft BASIC isn't a compiler; however, because it runs in the 16-bit world of GS/OS instead of the 8-bit world of ProDOS 8 (which is where Applesoft BASIC runs), you get a modest speed increase. According to Byte Works tests, GSoft BASIC programs run about 1.3 to 1.7 times faster than the same program running in

Applesoft, and my own testing seems to indicate that this is about right.

### **BASIC, meet the Toolbox**

As advertised, it's easy to port Applesoft BASIC programs to GSoft BASIC—in fact, you can load them directly within the GSoft environment—but it also has several new features, including complete and easy support for the Apple IIGS Toolbox.

If you're even moderately experienced at Apple IIGS programming, you know that using the Toolbox involves records containing fields of varying types of data, and pointers to data. These two key ingredients don't exist in any standard form of BASIC; GSoft BASIC adds easy-to-use records and pointers, and this is an important first step on the road toward full Toolbox capability.

The other key step is actual support for issuing Toolbox calls; GSoft does this admirably; a complete interface file is provided for calling every standard System 6.0.1 Toolbox function. In addition, GSoft includes commands for loading and unloading user tool sets, and lets you call their functions. This lets you call assembly language code from your BASIC programs by creating your own user tools.

### **What you get**

When your copy of GSoft BASIC arrives, you'll find two disks, a printed manual, and a product



***A new way of looking at things: GSoft BASIC is a 16-bit approach to a classic programming language.***

registration card. The first disk (":GSoft") has Apple's Installer on it, and will install GSoft BASIC on the disk of your choosing.

GSoft BASIC's system requirements are modest: 1.125 MB of memory on a ROM 3 system, or 1.25 MB on a ROM 01 system, and at least two disk drives, one of which must be a 3.5" drive so you can install the software. GSoft BASIC requires System 5.0.4, although System 6.0.1 is recommended.

There are three ways you can use GSoft BASIC. There's an ORCA/Shell language, BASIC, which lets you run GSoft programs from the ORCA/Shell. Since the ORCA/Shell doesn't come with GSoft BASIC, if you want to use this version of GSoft, you'll have to have one of the ORCA languages (ORCA/M, ORCA/

# Review: GSoft BASIC

C, ORCA/Pascal, or ORCA/Modula-2).

There's also the GSoft BASIC environment, which feels a lot like the BASIC.SYSTEM environment under ProDOS 8. This is entered by launching the GSoft.Sys16 application. Most of the commands are familiar, such as CATALOG for getting a list of files, PREFIX for changing the current working directory, and so forth, but there are some additional commands as well, such as COPY, for copying files.

The third way to run GSoft BASIC programs is one of the most innovative features of the environment: using the MakeRuntime utility provided with GSoft BASIC, you can create a Finder-launchable version of your BASIC program, as an Apple IIGS application. The application isn't compiled, but is a stand-alone GSoft BASIC interpreter with your BASIC program embedded inside it.

The applications you create using MakeRuntime are freely distributable; you don't have to pay royalties to the Byte Works; the only restriction is that you need to include a brief copyright notice indicating that Byte Works code is present in your application, just as is the case with all the ORCA products.

The manual is very good. It's not a tutorial (the Byte Works is considering creating a Learn to Program in GSoft BASIC tutorial package, and possibly a Learn to Program the Toolbox in GSoft BASIC package as well, depending on customer interest), but it serves as a good reference for people that already know BASIC.

The manual includes a quick-reference section to GSoft commands, an appendix describing

how to port Applesoft programs to GSoft BASIC, and information on creating user tool sets for use with GSoft BASIC.

If you don't already have BASIC experience, you'll probably want another book on BASIC programming. A book on Applesoft BASIC would be helpful, but you may find that a book on Microsoft BASIC is more useful; GSoft is more like Microsoft BASIC than Applesoft in many respects. A book on GW-BASIC for MS-DOS would work well.

## Differences from Applesoft

One area in which GSoft BASIC and Applesoft differ is in file I/O. Applesoft doesn't have any; both DOS 3.3 and ProDOS provided hacks that let you do file I/O by using the PRINT CHR\$(4) method. This doesn't work in GSoft. Instead, there are real, honest-to-goodness commands for opening, closing, and reading from and writing to files, and they work just like the standard Microsoft BASIC ones, which will help you port BASIC code from one platform to another.

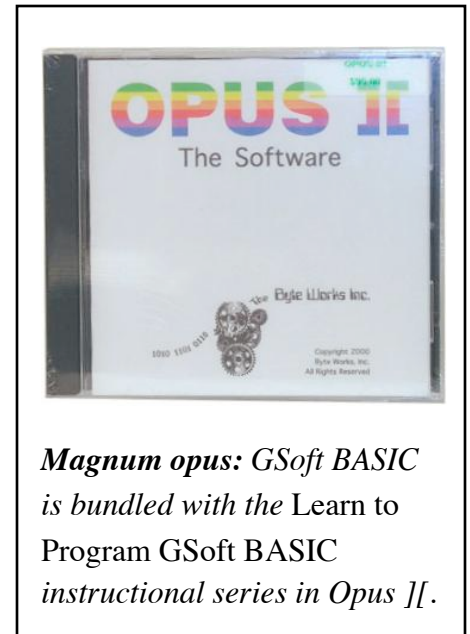
Another noticeable difference is that GSoft BASIC doesn't require that every line of code have a line number; they're only necessary on lines that are the target of a GOTO or GOSUB command.

There are several new data types. The complete list of simple types supported by GSoft BASIC is:

BYTE—Single-byte integer (0 to 255).

INTEGER—Two-byte integer (-32768 to 32767).

LONG—Four-byte integer (-2147483648 to 2147483647).



*Magnum opus: GSoft BASIC is bundled with the Learn to Program GSoft BASIC instructional series in Opus II.*

SINGLE—Four-byte single-precision floating-point number.

DOUBLE—Eight-byte double-precision floating-point number.

STRING—ASCII text string, up to 32767 characters.

You can create arrays of any of these, in as many dimensions as you like. The only restrictions are that any one array is limited to a maximum of 64k of memory, and you may only have 32767 subscripts per array.

There are two advanced types: the pointer and the record. A pointer is, simply, a pointer to a variable of one of the other types (or to a record).

A record is a collection of data of other types. For example, a mailing list database might have an array of objects of type ADDRESS:

```
TYPE ADDRESS
    NAME$
    STREET$
    CITY$
```

---

# Review: GSoft BASIC

```
STATE$  
ZIP$  
END TYPE
```

This establishes the ADDRESS type, which is a record containing the fields NAME\$, STREET\$, and so forth.

GSoft BASIC also provides commands for easily using the super high-resolution screen. The Applesoft HPlot and HColor= commands work on the SHR screen instead of the old high-resolution display. GSoft doesn't support low-resolution graphics.

Other exciting new features include a greatly-improved IF-THEN-ELSE over Applesoft, advanced looping (including WHILE-WEND and DO-LOOP), and real support for functions and subroutines.

Also included are two user tool sets, one for reading game controls, and another for reading date and time information from the Apple IIGS system clock and converting date and time information into strings.

There are other new commands as well—far too many to cover in this review.

## The GSoft BASIC shell

The GSoft shell (not to be confused with the more advanced ORCA/Shell) is roughly equivalent to the command line interface in Applesoft BASIC. It lets you execute BASIC commands in immediate mode; for instance, if you type:

```
PRINT "Hello world!"
```

The text "Hello world!" is immediately printed back out at you.

You also get the aforementioned CATALOG, COPY, and so forth, for managing your files.

You can enter lines of code into your GSoft program by typing them at the command line, starting with a line number (just like in Applesoft), or you can type "EDIT" and enter the full-screen editor, which is a more convenient way to edit your program code.

If you choose to use the command line to enter your program code, you must use line numbers on every line of code. If a single line of code exists without a line number, you have to use the full-screen editor.

The full-screen editor is the ORCA Editor, without any features changed or removed, so it's quite powerful. In fact, this is actually a small minus, as it's too powerful in some ways, and it's possible to get yourself into a mess by not saving your BASIC program in the place that GSoft expects it to be.

Once you've typed in your code, you just exit to the GSoft shell command line and type "RUN" to run your program.

The "LOAD" and "SAVE" commands work as usual to load programs from disk and save them back to disk. Also provided is a TSAVE command for saving your source code as a text file—this is useful if you need to port your code to another BASIC—as well as an SSAVE command, which saves the code into an ORCA GSoft BASIC source file, suitable for using with the ORCA version of GSoft BASIC.

## What's wrong with GSoft BASIC?

Not much. The editor is perhaps a little too powerful for GSoft BASIC's target audience, but that's

hardly a reason to frown upon the software.

There is a known bug in the currently-available version of GSoft BASIC which prevents you from writing desktop applications at this point in time—resources in the BASIC program's resource fork can't be loaded if you use StartUpTools to start the Resource Manager. A fix has been promised and should be along soon.

## The last word

GSoft BASIC is an exciting new product. For the first time ever, even beginning programmers can easily step right in and create real Apple IIGS software. The language is easy to learn for first-time programmers, familiar to longtime Applesoft programmers, and advanced enough that professional programmers won't scoff at it. Indeed, several serious Apple II programmers already have plans to develop real software using GSoft BASIC.

If you're looking for a way to learn to program, or to get real software written fast, give GSoft BASIC a try.

GSoft BASIC is \$40 plus \$5 shipping and handling for the original disk and book set, or \$30 for a digital-only downloadable edition. It is also available on the Opus II compilation, which is \$95 for the CD or \$80 for the download. All these Byte Works products are available from *Juiced.GS*:

**<https://juiced.gs/store/>**

***Eric Shepherd, a IIGS programmer who has created such titles as Shifty List, WebWorks GS and Wolfenstein 3D, is a technical writer for Be, Inc. in California's Silicon Valley.***

# On track with GSoft BASIC

## *Learning the language*

Before we get into it, let me take a moment to explain how this series is going to work. Instead of just starting at the beginning and going through linearly until you have a thorough understanding of GSoft BASIC, I'm taking an unusual tack. This will be a two-track series: Each article will be divided roughly in half.

The first half will be for beginning BASIC programmers, and will serve as an introduction to BASIC programming in general (and GSoft BASIC programming in particular). The second half will be for people who already know Applesoft BASIC and just want to learn the new features in GSoft BASIC. Beginning programmers should find that once they've finished the "Beginner's Track," they'll be ready to start back at the first article, in the "Advanced Track."

Even if you're already a BASIC programmer, you might want to browse the Beginner's Track, since it will introduce some basic programming concepts that are specific to GSoft BASIC. Note that the Advanced Track will occasionally refer to materials covered in later issues of the Beginner's Track. This will only happen with material that people who already know BASIC already know about.

This time, the Beginner's Track is going to cover the very basics of BASIC. The Advanced Track covers features more specific to GSoft BASIC, including new data types, records, and using the Toolbox.

## Beginners' Track

### HELLO WORLD

Start up GSoft BASIC. Once the prompt appears, type:

```
NEW
```

(Don't forget to hit the RETURN key). This tells GSoft BASIC to forget any code you've put in previously—no, you haven't done anything yet, so this isn't strictly necessary, but it's a good habit to get into: Clear everything out before you start a new program.

Now type:

```
EDIT
```

This gets you into the editor, where we'll do our actual programming. You should take a few minutes to read through the GSoft BASIC manual's chapter 6, which covers the editor.

Let's write the traditional first program anyone writes, which is cleverly entitled "Hello world." Type the following into the editor:

```
PRINT "Hello world!"  
END
```

The capitalization doesn't matter (GSoft BASIC will convert everything that's not in quotes into uppercase anyway, unless you're using the ORCA/Shell version). But otherwise you should type this in exactly as you see it above, including a RETURN at the end of each line.

Now press Apple-Q to quit the editor back to the GSoft BASIC ">" prompt (hit RETURN at the "Do you want to save" prompt). Type:

```
LIST
```

This command lists the contents of the current program you're working on. You should see the above lines spit back at you, except in all caps, and with some indentation added. GSoft BASIC automatically reformats your code into this format for you.

Now type:

```
RUN
```

This command runs the program you're currently working on. You should see the following results:

```
Hello world!
```

If you get an error message, you've made a typing error, and you should go back into the editor (type "EDIT" again) and make sure the two lines of code match exactly with the code above.



---

# Programming: GSoft BASIC, part 1 of 6

Let's look at that code again:

```
PRINT "Hello world!"
END
```

This program consists of two lines of code. The first uses the PRINT statement. This is used to print text to the screen. In this case, we're telling it to print the text "Hello world!". Note that the text is enclosed in quotes; this is important—without the quotation marks, GSoft BASIC will think that each word in the text is a variable or command name, and you'll get very unexpected behavior (probably the program will stop and present an error message).

The second line uses the END statement. This command simply tells GSoft BASIC that the program should stop (without an error message) at that point.

## GETTING TO KNOW YOU

Let's try expanding Hello World to ask for the user's name, and use it in the greeting message. Go back into the editor, and change the program to look like this:

```
INPUT "What's your name? ";NAME$
PRINT "Hello, ";NAME$;"."
END
```

Now quit the editor and RUN this program. You should see:

```
What's your name?
```

With a cursor flashing after the question mark. Type your name and hit RETURN. You should then see:

```
Hello, Eric.
```

(Or whatever name you typed). Again, if an error occurs, go back to the code and check for mistakes.

As you may already have guessed, the INPUT command asks the user to input something from the keyboard. It displays a prompt (in this case, "What's your name?"), then waits for the user to type in a response, followed by the RETURN key.

NAME\$ is a variable. A variable is a container, of sorts, that holds information (in this case, information typed in by the user, but it can also be generated by other program code, which we'll see shortly). Variables can be of different types; each type is designed to hold a different kind of information. The dollar sign (\$) at the end of the variable NAME\$ indicates that this is a

string variable. A string contains ASCII text data—up to 32K worth!

Note that you can't give a variable the same name as a BASIC command; for example, you can't have a variable named "PRINT" because that's the name of a BASIC command.

So the INPUT statement asks the user to type in their name, and saves the input text into the string variable NAME\$.

The PRINT statement makes another appearance here, this time in a somewhat more complicated form:

```
PRINT "Hello, ";NAME$;"."
```

Let's look at this one chunk at a time. As you recall, text contained in quotes is treated as a literal string (which means that it's treated as text, just like you wrote in the code). So this PRINT statement begins by printing "Hello, " to the screen.

Next comes a semicolon (;). The semicolon is used to separate different pieces of information that a PRINT statement is to print. This is followed by the variable NAME\$—the contents of this variable are printed next. Then there's another semicolon, followed by ".". This causes the desired output.

Experiment with the format of the PRINT statement in this program and see if you can change the output message. You can string together as many things separated by semicolons as you like, including literal strings (such as "Hello") and any kind of variable, such as string variables as we've seen above.

## DOING MATH

Let's try making a program to add numbers together. First, let's start simple:

```
PRINT 5 + 10
END
```

RUN this program. The number 15 should be displayed. That's obviously the right answer. In this case, we're using the PRINT statement to print the results of a mathematical expression, "5 + 10" to the screen.

Let's spice this up a tad:

```
PRINT "5 plus 10 is ";5 + 10;"."
END
```



# Programming: GSoft BASIC, part 1 of 6

RUN this program. You'll see the text "5 plus 10 is 15." You can embed expressions into the PRINT statement this way. A numeric expression is any syntactically-correct sequence of numbers, operations, and functions that evaluates to a number (there are also string expressions, which we'll play with later). All the following are examples of numeric expressions:

```
5
2+3
2*(10+3)
(5)
-5+1
NUMBER*5+X
```

In the last of these, we see that variables (such as NUMBER and X) can be used in numeric expressions.

The following are examples of invalid expressions; they contain syntax errors that will cause your program to break with a syntax error:

```
+
(5*2
5**2
*2+4
```

These should be wrong for fairly obvious reasons: in the first case, there aren't any numbers, so this can't be evaluated. The second example is missing a close parenthesis ")". The next one has two multiplication signs in a row, and the last one has a multiplication sign at the beginning of the expression, which doesn't make any sense.

Back to our math practice program. How about making it so the user can input the numbers to be added? Try this program:

```
INPUT "First number: ";NUMBER1
INPUT "Second number: ";NUMBER2
PRINT NUMBER1;" plus";NUMBER2;" is
";NUMBER1 + NUMBER2;". "
END
```

N1 and N2 are single-precision floating-point numbers (this is the default data type for any variable that you don't explicitly specify a type for—we'll look at how to specify data types in a moment). The two INPUT statements ask the user to input two numbers; these values are placed in the variables N1 and N2.

The PRINT statement then prints out the results. For example, if the user types 1.5 and 3 for the two numbers, the output would be:

1.5 plus 3 is 4.5.

Mathematical computations are done very much like they are in algebra. Let's take a look at some of the basic operations.

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	^

You can also group expressions using parentheses; the standard rules of mathematics apply:

```
PRINT 2*(5+5)
20
PRINT 10-5*2
0
PRINT (10-5)*2
10
```

There are also mathematical functions for performing operations like ABS(), which returns the absolute value of an expression, or COS(), which returns the cosine of an expression. For example:

```
PRINT COS(3.14)
-0.9999987
PRINT ABS(COS(3.14))
0.9999987
```

The first example prints the cosine of pi. This doesn't come out to exactly -1 because we're not being very precise: pi isn't exactly 3.14. On top of that, single-precision floating-point numbers aren't very accurate either. We'll look at ways to improve this later.

Check out the GSoft BASIC manual's quick reference of functions, which starts on page 313. Play with the numeric functions there for a while.

Before going on, try experimenting with these operations for a while. Keep in mind that you can try them by typing PRINT statements right at the GSoft BASIC "]" prompt (this is called executing commands in immediate mode).

---

# Programming: GSoft BASIC, part 1 of 6

## DATA TYPES

GSoft BASIC supports five numeric data formats: single-precision floating-point, double-precision floating-point, byte, integer, and long integer. Let's look at each of them:

The byte data type can store values between 0 and 255. You specify a byte variable by using the ~ character after the variable's name, such as BVALUE~.

The integer data type can store values between -32768 and 32767. Variables ending with the character % are treated as integers, like INTVALUE%.

Long integers can store values between -2147483648 and 2147483647, and are represented by variables with the character & at the end of their name, such as LVALUE&.

Single-precision floating-point values use four bytes of memory, and can have an exponent from 1e-38 to 1e38, and are accurate to seven decimal digits. Variables with no character after their name, or with the ! character after their name, are treated as single-precision real numbers.

Double-precision real numbers use eight bytes of memory, and are accurate to 15 decimal digits, with an exponent that can range from 1e-308 to 1e308. Identifiers ending with the # character are treated as real numbers.

There are two special values that floating-point computations can return: INF and NaN. These represent infinity and "not a number," respectively. INF can be returned if a value exceeds the maximum value supported by a data type, and NaN can be returned if the result of a calculation is illegal.

For instance, try:

```
A = 1E50
PRINT ABS(A)
```

The result is "inf". That's because 1E50 is outside the range of values supported by single-precision numbers.

Try this:

```
PRINT SQR(-1)
```

The result is "NaN". That's because the square root operation, SQR(), is undefined for negative numbers (GSoft BASIC doesn't handle complex numbers).

I think that's enough for the beginner's track for this time. We've covered a lot of ground already. Experiment. Look through the GSoft BASIC manual and see if you can figure anything else out on your own.

Next issue, we'll start looking at ways to create programs that make decisions and perform tasks repeatedly.

## Advanced Track

### DATA TYPES REVISITED

BASIC supports several types of data, including bytes, integers, long integers, strings, and floating-point numbers (both single- and double-precision). As covered in the first lesson of the Beginner's Track (above), these variables are established, by default, by appending a special character to the end of the variable name.

For example, a string has a "\$" at the end of the name, such as NAME\$.

Most other languages, such as C or Pascal, require you to declare your variables in advance before you use them. GSoft BASIC's default behavior is to create variables for you as you use them. This can be convenient for beginning programmers, but as you build larger and more complex projects, it can get confusing.

GSoft BASIC lets you declare variables in advance by using an advanced form of the DIM statement (this statement will be covered in part 3 of the Beginner's Track). Normally used to create arrays, DIM doubles as the statement to declare variables in advance.

For instance, if you want an integer variable named ID, you can declare it as follows:

```
dim ID as Integer
```

This establishes that the variable ID represents an integer value. Note that we don't need the "%" character to identify it as an integer; the DIM statement establishes this type for us, so the special character isn't needed. You can do this with any type of variable:

```
dim Name as String
dim Cash as Single
dim Key as Byte
```

You can define your own data types in GSoft BASIC by using the TYPE statement. There are two uses for this ability: first, you can use it to create specialized

---

# Programming: GSoft BASIC, part 1 of 6

names for existing types, to help you manage your software development more easily, and second, you can create record types.

Let's say you're working on a program that computes the acceleration required to change an object's speed in a given amount of time. Here's a program that will do the trick:

```
!
! Compute the acceleration necessary to
change from
! one velocity to another in a given
amount of time.
!
type accelType as Single
type velocityType as Single
type timeType as Single
dim a as accelType
dim v1 as velocityType
dim v2 as velocityType
dim t as timeType
! Explain the program
print "This program computes the
acceleration necessary to change"
print "from one velocity to another in a
given amount of time."
print
! Get input
input "Enter the object's current
velocity (in m/s): ";v1
input "Enter the object's desired
velocity (in m/s): ";v2
input "Enter the time allowed for the
change (in s): ";t
! Do the math
a = (v2 - v1) / t
print "The required acceleration is
";a;" m/s^2."
end
```

Note that we create new data types: `accelType`, `velocityType`, and `timeType`, which we use for creating our variables for acceleration, velocity, and time. They're defined as `Single`; single-precision floating-point numbers.

If you use this program to compute the force exerted on a 5 kg object being accelerated by the Earth's gravity ( $9.8 \text{ m/sec}^2$ ), you get the following output:

This program computes the acceleration necessary to change from one velocity to another in a given amount of time.

```
Enter the object's current velocity (in
m/s): 0
Enter the object's desired velocity (in
m/s): 50
Enter the time allowed for the change
(in s): 4.1
The required acceleration is 12.19512 m/
s^2.
```

But let's say you start using this program, then realize that single-precision floating point numbers just aren't accurate enough for you. Because we created custom types for these, instead of directly using type `Single`, we can fix the types easily by changing the type definitions to:

```
type accelType as Double
type velocityType as Double
type timeType as Double
```

Now when we run the program again, the math is done using double-precision floating-point math, and we get a different result for the same inputs:

```
The required acceleration is
12.19512223488 m/s^2.
```

As you can probably guess, the more complicated the program, the more benefit you get from using custom data types in this manner—especially when your program has subroutines defining local variables for use in calculations.

## RECORDS

Most modern programming languages support the concept of a record (in C, they're called structures). A record is a group of data fields of various types that are treated as a single data object.

For instance, a record describing an entry in an address book might have fields like "name," "address," "city," "state," "zip code," "age," and "phone number."

GSoft BASIC lets you create records. For instance, the address book record might look something like this:

```
type address
  Nombre as String
  Address as String
  City as String
  State as String
  Zip as String
```

---

# Programming: GSoft BASIC, part 1 of 6

```
Age as Integer
Phone as String
end type
```

Note that the Zip Code and Phone Number are being defined as strings, not integers, because they can contain characters other than digits, such as dashes and parentheses. Also, the "Name" field is called "Nombre" here because the word NAME is reserved by GSoft BASIC.

If you want to use an address record in your code, you declare an address variable using the DIM statement as usual:

```
dim Joe as address
```

You can then assign values to the fields in the record:

```
Joe.Nombre = "Joe Smith"
Joe.Address = "920 Evergreen Terrace"
Joe.City = "Springfield"
Joe.State = "ST"
Joe.Zip = "29371"
Joe.Age = 31
Joe.Phone = "(810) 555-1937"
```

Each field can be treated just like any variable; they can be used in expressions as usual. The following code will print an address label for Joe:

```
dim addressLine as String
addressLine = Joe.City + ", " +
Joe.State + " " + Joe.Zip
print Joe.Nombre
print Joe.Address
print addressLine
```

This will output:

```
Joe Smith
920 Evergreen Terrace
Springfield, ST 29371
```

Experiment with records. Try making a version of this program that will use the INPUT statement to input data and fill out the record. If you're coming back to this lesson after finishing the entire Beginner's Track (which hasn't been written yet, of course), try creating an array of records and providing a simple interface for letting the user choose records to view and edit. Don't

forget you'll need to keep a counter variable to track how many records have been used in your array.

## TRUE SUBROUTINES

The classic BASIC method of providing subroutines by using the GOSUB and RETURN statements doesn't provide a method of passing data into the subroutine and receiving data back; this relies on global variables to do the job.

Modern structured programming techniques mandate subroutines with local variables; these are variables that only have meaning within the subroutine, and don't affect the world outside.

GSoft BASIC provides a means for creating and using subroutines of this nature. Let's have a look at a version of our acceleration program, changed to do the calculation and printing in a subroutine:

```
!
! Compute the acceleration necessary to
change from
! one velocity to another in a given
amount of time.
! Uses a function to do the math.
!
type accelType as Double
type velocityType as Double
type timeType as Double
dim v1 as velocityType
dim v2 as velocityType
dim t as timeType
! Explain the program
print "This program computes the
acceleration necessary to change"
print "from one velocity to another in a
given amount of time."
print
! Get input
input "Enter the object's current
velocity (in m/s): ";v1
input "Enter the object's desired
velocity (in m/s): ";v2
input "Enter the time allowed for the
change (in s): ";t
! Use the subroutine to print the
result.
call ComputeAcceleration(v2, v1, t)
end
!
```



---

# Programming: GSoft BASIC, part 1 of 6

```
! Return the acceleration, given the
velocities
! and the time.
!
sub ComputeAcceleration(v2 as
velocityType, v1 as velocityType, t as
timeType)
    dim a as accelType
    a = (v2 - v1) / t
    print "The required acceleration is
";a;" m/s^2."
end sub
```

The main program has the line:

```
call ComputeAcceleration(v2, v1, t)
```

This passes control to the subroutine ComputeAcceleration, passing the variables v2, v1, and t as parameters.

Look at ComputeAcceleration next. The parameter list is formatted similarly to the DIM statements we use to declare variables. The parameters are named v2, v1, and t, and are of the types velocityType, velocityType, and timeType respectively.

These have the same names as the global variables we're using for the same purposes, but that's not a requirement. The subroutine would work just as well like this:

```
sub ComputeAcceleration(endV as
velocityType, startV as velocityType, t as
timeType)
    dim a as accelType
    a = (endV - startV) / t
    print "The required acceleration is
";a;" m/s^2."
end sub
```

If the code in the subroutine changes the value of one of the parameters, the corresponding variable in the main program is also changed; this is called "passing parameters by reference" because the variable itself is passed, instead of the variable's value. For example, try the following code:

```
dim num as Integer
num = 10
print "Before: ";num
call Display(num)
print "After: ";num
```

```
end
sub Display(v as Integer)
    v = v/2
    print "Display says ";v
end sub
```

This program, if you type it in and RUN it, presents the following output:

```
Before: 10
Display says 5
After: 5
```

Note that the value in the variable num is changed by Display. If this behavior isn't what you want, you can force "passing by value" by enclosing the variables you pass into subroutines in parentheses, like this:

```
call Display((num))
```

If you make that change then run the program again, you get the following output:

```
Before: 10
Display says 5
After: 10
```

Passing by value means that the actual value is passed, instead of the variable, so it can't be changed by the subroutine. If you pass a number or numeric expression, that's passed by value. Try:

```
call Display(num+3)
The output is now:
Before: 10
Display says 6
After: 101
```

GSoft BASIC also supports functions. A function is a special kind of subroutine that returns a value. Let's return to our physics problem above and redo it to use a function to do the math:

```
function ComputeAcceleration(v2 as
velocityType, v1 as velocityType, t as
timeType) as accelType
    dim delta as velocityType
    delta = v2 - v1
    ComputeAcceleration = delta / t
end function
```

This function computes the result of the formula  $a = (v1-v2)/t$  and returns the result. The return value is

---

# Programming: GSoft BASIC, part 1 of 6

specified by assigning a value to the function name itself, like this:

```
ComputeAcceleration = delta / t
```

You use your functions just like the built-in functions in GSoft BASIC. For instance, you can use the ComputeAcceleration function in any of the following ways:

```
print "The required acceleration is  
";ComputeAcceleration(v2, v1, t);" m/s^2."
```

or...

```
a = ComputeAcceleration(v2, v1, t)
```

You can use classic, BASIC-style variables in your subroutines and functions, if you want to:

```
function ComputeAcceleration(v2#, v1#,  
t#) as #  
    ComputeAcceleration = (v2# - v1#)/t#  
end function
```

However, throughout this course I'll be using the modern variable style because it's easier to read.

Subroutines and functions don't have to accept input parameters. For example:

```
sub Error  
    print "An error has occurred!"  
end sub
```

or...

```
function CheckMouse as Boolean  
    if Button(0) then  
        CheckMouse = true  
    else  
        CheckMouse = false  
    end if  
end function
```

The latter function calls the Toolbox function Button() to read the state of the mouse button (note this won't work unless you've started the Event Manager). It then returns true if the button is down, otherwise it returns false.

If your subroutine or function needs to access variables in the main program, you could just pass them all into the function, but that can be very tedious and

hard to keep track of what all is going on. Instead, you can use the SHARED statement:

```
dim v as Integer  
v = 10  
sub Calculate  
    shared v  
    print v * 5  
end sub
```

The Calculate subroutine prints out the value of the global variable v times five. If you left out the SHARED statement, this would print zero, because the local variable v is created automatically with a default value of 0.

Which brings up the final note for this time: beginning with GSoft BASIC 1.1, you can use a special command to tell GSoft BASIC to produce an error if you use a variable you haven't declared using the DIM statement, instead of just creating it on-the-fly with a value of zero.

```
PRAGMA AUTODIM = OFF
```

You can turn auto-dimensioning of variables back on by using the statement:

```
PRAGMA AUTODIM = ON
```

This can be very helpful: it's very easy to miss problems where you've forgotten to declare a variable, or forgot to use the SHARED statement in a function or subroutine. Without this command, you won't get an error message, but the code won't work right. With the command, you'll get an "Undefined variable" error. This is a very useful debugging tool. For example, let's consider the last example again, without the SHARED statement:

```
dim v as Integer  
v = 10  
sub Calculate  
    print v * 5  
end sub
```

This printed 0 before, because v is autodimensioned and given the default value 0. If you add the line "pragma autodim = off" at the beginning of the program, you'll get an error on the line "print v \* 5", and you can figure out fairly easily what's wrong and fix the problem.

Next, we'll look at how to do I/O, including disk files and more advanced text-formatting features.

# Digging deeper into GSoft

## *New lessons for novices and old hands*

### Beginners' Track

#### COMMENTS

Something that can be very useful as you're writing code is the ability to add comments to your program. These don't actually affect the operation of your code, but help to remind you how your code works. GSoft BASIC provides two forms of comments. Let's take a look:

```
PRINT "This is code."
REM This is a comment.
! This is a comment, too.
PRINT "This will print to the
screen." : ! Yet another comment
```

The REM (short for remark) statement can be abbreviated "!", as shown above.

Note on the last line the use of the colon (:) character. This lets you put multiple statements on a single line. You could even do this:

```
PRINT "Hello "; : PRINT "world" : REM
This prints "Hello world"
```

However, once a comment has been used on a line, you can't put any other code behind it. For example:

```
PRINT "This is code." : REM This is a
comment : PRINT "Is this code?"
```

Everything after the REM keyword is treated as a comment, including the intended PRINT statement.

#### FORMATTING TEXT OUTPUT, PART 1

Consider again this example, which demonstrates another useful BASIC programming feature we skimmed over last time:

```
PRINT "Hello "; : PRINT "world" : REM
This prints "Hello world"
```

Note the semicolon (;) character after the closing quote in the first PRINT statement. The semicolon, used

in a PRINT statement, tells GSoft BASIC not to insert a carriage return after that line. In other words, the printing cursor remains immediately after the last character printed. Last time, we used the semicolon to combine literal strings and variables in a single line of output. Try this yourself, using the following statements:

```
PRINT "This " : PRINT "is" : PRINT " a
test."
PRINT "This "; : PRINT "is"; : PRINT "a
test."
PRINT "This "; "is"; " a test."
A = 5 : PRINT "The value is "; A; "."
```

The first example prints:

```
This
is
a test.
```

The second and third examples both print:

```
This is a test.
```

The last example prints:

```
The value is 5.
```

You can also do some basic column layout, if you want to display tables of information, by using the comma character (,):

```
PRINT "Number", "Value"
PRINT "1", "5.2"
PRINT "2", "7.1"
PRINT "3", "9.2"
```

This will print:

Number	Value
1	5.2
2	7.1
3	9.2

This issue's Advanced Track covers more powerful print formatting.

---

# Programming: GSoft BASIC, part 2 of 6

## BRANCHING OUT

Sometimes a program needs to jump around from one line of code to another elsewhere in the program, instead of just running straight through. The simplest method for doing this is the GOTO statement. This statement has generated a great deal of controversy; its use is generally considered poor programming style because using GOTO too much can make your code hard to understand. However, there are cases in which it's quite useful.

Here's an example:

```
10 print "Hello world!"
20 goto 10
```

This two-line program prints "Hello world!" over and over again, until you press Control-C to break out of the program. Notice how each line starts with a number? Those are called line numbers. Applesoft BASIC requires them on every line, but they're optional in GSoft. The GOTO statement requires that the line that you want to jump to has a line number on it.

You could write this program like this:

```
10 print "Hello world!"
goto 10
```

As you can see, you can mix lines with and without line numbers in the same program. The GOTO statement redirects program execution so that the next statement that gets processed is the first statement on the line you specify (you can't jump to the second statement on the same line of code).

This is the simplest type of loop you can create in GSoft BASIC. A "loop" is a piece of code that runs more than once (either infinitely or until some condition arises). We'll look at more controlled looping next time.

Starting with GSoft BASIC 1.2 (which is available now), you can use named labels instead of line numbers, like this:

```
top:
    print "Hello world!"
    goto top
```

The label can't be the same as a GSoft BASIC reserved word (you can't have a label called PRINT or GET, for example).

## MAKING DECISIONS

Even with the ability to accept user input, there's not a lot you can do if you can't make decisions based on what the user types at the keyboard. BASIC provides two primary ways of making decisions based on a condition: the first is the IF/ELSE combination of statements, and the other is the SELECT CASE statement. We'll look at SELECT CASE later; this issue, we're going to focus on IF/ELSE.

Let's look at an example. Here we're asking the user to enter a percentage value and stash it in the variable p:

```
input "Type the percentage: ";p
if p > 100 then print "Greater than
100%."
```

The second line checks to see if the number, p, is greater than 100. If it is, the message "Greater than 100." is printed; if the number is less than 100, nothing happens. This does just what it looks like; it's almost like reading English.

The format of the IF-THEN statement is:

```
IF expression THEN statement
```

If the expression is true, then the statement is run; otherwise it's not. You can use any expression that evaluates to true or false (this is called a Boolean expression). Some examples include:

```
if A$ = "CODE" then print "True"
if (A/2)+3 >= 15 then print "True"
if abs(a) = 10 then goto 500
```

You might have noticed the interesting operator ">=" in the second example above. This is "greater than or equal to." Likewise, less than or equal to can be written as "<=". You can reverse the equal sign and the greater/less than sign ("=>" and "<=") if you like; it doesn't matter. The "not equal to" operator is "<>".

But let's say we also need to display a warning message when the value is greater than 100. The following code will accomplish this:

```
input "Type the percentage: ";p
if p > 100 then print "Too high!
Setting to 100%." : p = 100
print p;"% entered."
```



---

# Programming: GSoft BASIC, part 2 of 6

If you run this and type the value 50, you'll see the output:

50% entered.

If you type 150, you'll see this output:

Too high! Setting to 100%.  
100% entered.

This pegs the maximum value the user can enter to 100% by forcing the value to 100% if it's too big (we'll cover the do's and don'ts of user-friendly software design later). This takes advantage of the fact that every statement on the same line as an IF statement, AFTER the IF statement, is executed if and only if the expression is true.

For example:

```
if 1 = 1 then print "A" : print "B" :  
print "C"
```

This will always print:

A  
B  
C

Because the expression `1 = 1` is always true. Interestingly, you get the same result if you use the line:

```
if 1 then print "A" : print "B" : print  
"C"
```

That's because the expression `1` has the value 1; any nonzero result is considered "true."

But what if the user types a value less than 0? That's not a valid percentage. We need to extend this code to handle that case:

```
input "Type the percentage: ";p  
if p > 100 then print "Too high!  
Setting to 100%." : p = 100  
if p < 0 then print "Too low! Setting  
to 0%." : p = 0  
print p;"% entered."
```

This code works, but putting multiple statements on a single line isn't very stylish, and can be confusing, especially when dealing with IF statements like these. Fortunately, GSoft BASIC supports structured IF blocks:

```
input "Type the percentage: ";p  
if p > 100 then  
    print "Too high! Setting to  
100%."  
    p = 100  
end if  
if p < 0 then  
    print "Too low! Setting to  
0%."  
    p = 0  
end if  
print p;"% entered."
```

This is especially valuable if you have a large number of statements that have to be executed as the result of an IF expression being true. It would be very cumbersome to have to put them all on the same line. This lets you organize your code more cleanly.

Be sure you remember to end your block IF statements with "END IF" though; otherwise GSoft won't know where the conditional code ends! Using indentation properly, like these examples do, can help keep you keep track of where your blocks begin and end.

Let's look at another situation. Let's say you're creating a menu of three options: "Add two numbers," "Subtract two numbers," and "Quit." The user will pick one of these options using a number key. Here's code that does it:

```
10 print "Main menu"  
    print "  1) Add 10 + 10"  
    print "  2) Subtract 20-10"  
    print "  3) Quit"  
    input "Choose (1-3): ";option  
    if option = 1 then print "10+10=";  
10+10  
    if option = 2 then print "20-10=";  
20-10  
    if option = 3 then end  
    ! Start back at the top again  
    print  
    goto 10
```

There are a few problems with this code as it stands. First, this code doesn't handle the error condition that occurs if the user types a negative number or a number greater than three. Also, as it stands, it compares the typed option number to all three possibilities (1, 2, and 3), instead of stopping when one matches. The code works, but it's not very tidy.

Here's code that fixes the problem:

---

# Programming: GSoft BASIC, part 2 of 6

```
top:
  print "Main menu"
  print "  1) Add 10 + 10"
  print "  2) Subtract 20-10"
  print "  3) Quit"
  input "Choose (1-3): ";option
  if option = 1 then
    print "10+10=";10+10
  else if option = 2 then
    print "20-10=";20-10
  else if option = 3 then
    end
  else
    print chr$(7);"*** You
must enter a number from 1-3!"
  end if
  ! Start back at the top again
  print
  goto top
```

This version uses the ELSE statement to make decisions more intelligently. Each comparison in the chain of IF / ELSE IF / ELSE blocks is made until one matches, then the program skips over all the remaining comparisons to just after the END IF statement. This is much more efficient, and is better programming style to boot.

If the user runs this program and types "2" as input, it gets compared to 1 (and doesn't match), so then it gets compared to 2. That matches, so the line:

```
print "20-10=";20-10
```

gets executed. Then control jumps down to the comment "! Start back at the top again".

If the user types 6, all three comparisons (to 1, 2, and 3) fail to match, so finally the catch-all "ELSE" block gets run, and the message "\*\*\* You must enter a number from 1-3!" gets printed.

The CHR\$( ) function takes an integer value and returns a string containing the character corresponding to that ASCII code. Character 7 is the bell character. Printing chr\$(7) causes the computer to beep.

Judicious use of IF / ELSE IF / ELSE blocks can make your program much more robust and keep the user from crashing it. (Remember, if it crashes, they'll blame you, even if they made the mistake that broke it!)

Next time we're going to look at arrays and some more advanced forms of looping through the same code

over and over again. Try playing with the concepts we covered this time. Next time, I'm going to start giving you specific projects to try.

## Advanced Track

### PRINTING IN THE '90s

GSoft provides an advanced form of the PRINT statement called PRINT USING. PRINT USING lets you specify a formatting template that tells PRINT how to lay out the text it prints (similar to how printf() works in C); this gives you powerful control over the appearance of printed text.

Let's look first at a sample program. This program prints a table of items being purchased, the quantity of each item, the cost per item, and the total cost:

```
dim item(5) as string
dim qty(5) as integer
dim cost(5) as single
dim total(5) as single
dim i as integer
! Set up the item names
for i = 1 to 5
  read item(i), qty(i), cost(i)
  total(i) = qty(i) * cost(i)
next
! Set up the display
home
print "Item"; spc(16); "Qty."; spc(4);
"Cost Each"; spc(4); "Total Price"
print "-----"; spc(4);
"----"; spc(4); "-----"; spc(4);
"-----"
! Print the table
for i = 1 to 5
  print using "\          \
####    $$$$##    $$$,###.##"; item(i),
qty(i), cost(i), total(i)
next
end
! Data. Each line is item name,
quantity, and cost, in
! that order.
data "gsAIM", 2, 5
data "WebWorks GS", 14, 20
data "ImageMaker", 4, 5
data "Shifty List", 7, 20
data "Value Pack", 28, 55
```

Arrays, READ, DATA, and FOR-NEXT loops are covered in the Beginner's Track of Part 3 of this series,

# Programming: GSoft BASIC, part 2 of 6

which will be in the next issue. The interesting part is the line:

```
print using "\          \ ####
$$$#.##    $$##,###.##"; item(i), qty(i),
cost(i), total(i)
```

This strange-looking construction formats the output from our table. Here's what the output looks like:

Item	Qty.	Cost Each	Total Price
-----	----	-----	-----
gsAIM	2	\$5.00	\$10.00
WebWorks GS	14	\$20.00	\$280.00
ImageMaker	4	\$5.00	\$20.00
Shifty List	7	\$20.00	\$140.00
Value Pack	28	\$55.00	\$1,540.00

Each character in a format field represents space for exactly one character to be substituted in the final printed string.

The "\ \" part indicates that a string will be printed in that space. The string is to be up to 18 characters long (that's how many characters the template takes up, including the backslashes and the spaces). If the string specified is too long, it's chopped off at 18 characters.

Then there are some spaces, to align the next field under the Qty. column. The format string here is "\$\$#.##". This indicates that the next value is a number, and should be padded, using spaces, to be four characters wide, right-justified in the column.

If a number is too big to fit in the allotted space, the entire number prints anyway; it doesn't get lopped off. Be aware of this when designing your software—you either need to be sure the formatting can handle any possible number, or you can design your display so it doesn't matter if a number takes up more than its fair share of space.

After this, there are more spaces for alignment purposes, then the format string "\$\$#.##". This causes a number to be formatted to fit an 8-character wide field, right justified as before. The difference is that no matter what the value is, the number will be printed with two decimal places, and a "\$" character will be placed just before the first character. This formats our dollar amounts nicely.

The last format substring is "\$\$##,###.##". This is similar to the last one; it displays a dollar amount with

two decimal places, in a column 11 characters wide. The comma indicates that if the number is greater than 1000, a comma should be placed before the thousands digit.

After the template string comes a list of the values to be inserted in place of the formatting information. These must be of the correct types to match the formats, or you'll get errors when you run the program.

If you want the dollar signs to be aligned in the first character of the dollar amount columns, replace the second "\$" character with a "#". This causes the following output:

Item	Qty.	Cost Each	Total Price
-----	----	-----	-----
gsAIM	2	\$ 5.00	\$ 10.00
WebWorks GS	14	\$ 20.00	\$ 280.00
ImageMaker	4	\$ 5.00	\$ 20.00
Shifty List	7	\$ 20.00	\$ 140.00
Value Pack	28	\$ 55.00	\$ 1,540.00

You could also change the dollar amount templates so that instead of padding with spaces, the "\*" character is used. This is done by using "\*\*\*" in the format string, like this:

```
print using "\          \ ####
$$$#.##    **$#,###.##"; item(i), qty(i),
cost(i), total(i)
```

This results in the following table:

Item	Qty.	Cost Each	Total Price
-----	----	-----	-----
gsAIM	2	***5.00	*****10.00
WebWorks GS	14	**20.00	*****280.00
ImageMaker	4	***5.00	*****20.00
Shifty List	7	**20.00	*****140.00
Value Pack	28	**55.00	**1,540.00

By default, no sign is displayed if the number is positive (a negative sign is displayed before the number if it's negative, however). If you want to force the sign to be displayed for every number you print, you can use the "+" character at the beginning of the format field:

```
print using "+##,###,###.##"; 5927123.1
```

This prints:

```
+5,927,122.90
```

---

# Programming: GSoft BASIC, part 2 of 6

You can also specify that you want the sign to be displayed after the digits, by putting a "+" or "-" character as the last character in the format. If you use "-", the sign will only be displayed if the number is negative; if you use "+", the sign will always be displayed:

```
print using "###,###,###.#+"; -54321.98
      54,321.9-
print using "###,###,###.#+"; 5927123.1
      5,927,122.9+
print using "###,###,###. #-"; -54321.98
      54,321.9-
print using "###,###,###. #-"; 5927123.1
      5,927,122.9
```

You can of course combine text and formatting together, like this:

```
sum = 0
for i= 1 to 5
    sum = sum + total(i)
next

print
print using "The total is $$#,###.##.
This is order _####."; sum, 592
```

This displays as:

```
The total is  $1,990.00.  This is order
#592.
```

Notice the "#" character in our output? This is because we used an underscore character ("\_") before the first "#" character in the order number formatting field. The underscore character tells GSoft BASIC to actually print the following formatting character, instead of treating it as a formatting character. If you want to print an underscore, you put two underscores in the text: "\_\_".

There are a few other formatting methods you can use; these are discussed in the GSoft BASIC manual, starting on page 169.

## INTRODUCTION TO FILE I/O

GSoft BASIC deviates substantially from the Applesoft disk I/O model. In Applesoft, you can only

access a single file at a time. GSoft BASIC supports up to eight open files at once; you assign a number between 1 and 32,767 to each file when you open it. This file number is then used to refer to the file until it's closed. Let's look at a version of "Hello world" that writes its output to a disk file instead of to the screen.

```
open "Hello.txt" for output as #1
print #1, "Hello world!"
close #1
```

The first line opens the file "Hello.txt" for output, and assigns it the file number 1. You could just as well have specified file number 32000, or 5, or 123.

The second line prints the text "Hello world" to file number 1. The last line closes file number 1. You should always close a file when you're done using it; this lets GS/OS clean up after itself (and ensures that cached data is written to the file).

This type of file is called a Sequential File. Data is written straight through from the beginning of the file to the end. There's another kind of file, called Random Access, which we'll look at next time.

You can write any kind of data you want to into a file. Let's write the table from the PRINT USING example above into the file:

```
open "table.txt" for output as #1
print #1, "Item"; spc(16); "Qty.";
spc(4); "Cost Each"; spc(4); "Total Price"
print #1, "-----"; spc(4);
"----"; spc(4); "-----"; spc(4); "----

! Print the table

for i = 1 to 5
    print #1 using "\
####  $####.##  **$,###.##"; item(i),
qty(i), cost(i), total(i)
next
close #1
```

The result is a text file containing the table from the last section.

You can print anything into the file:

```
i = 50
print #1,i
print #1,"This is the number ";50;"."
```

These will work fine.



---

# Programming: GSoft BASIC, part 2 of 6

You can append to an existing file (without deleting the data already present) by specifying "FOR APPEND" like this:

```
open "table.txt" for append as #1
```

If you want to read data from a text file, you would open the file FOR INPUT, like this:

```
dim filename as string
dim s as string

input "Filename: ";filename

open filename for input as #1
while not eof(1)
    line input #1, s
    print s
wend
close #1
```

This program asks for a filename, then opens it for input, giving it file number 1.

Then a while loop is used. The eof() function returns true when the specified file number has no data left to read (the end of the file has been reached).

This loop, then, repeats until the end of the file is reached. We repeatedly read lines of text from the file using LINE INPUT. The LINE INPUT statement is just like INPUT, but doesn't display a prompt, and allows commas and quotation marks in the text.

Each line is then printed using the PRINT statement.

When the loop is finished, the file is closed using the CLOSE statement as before.

You can of course read numeric values; anything you can print to a file can be input back in; however, note that the file actually contains ASCII text data—we'll look at reading and writing binary data later.

```
open "data" for input as #1
input "";num
close #1
```

This will read the first line from the file "data" as an integer value. If the first line is "50", num will be set to 50. If the first line isn't an integer value, an error "Number expected: Reenter" will occur.

We'll continue to look into file I/O next time.

---

## Getting down to the nitty gritty

Hello again! Our exploration of GSoft BASIC continues into another month! This month's beginner's track will examine arrays and looping. The advanced track will look at some more file access and pointers.

Due to a deadline I have in my real-world job, this episode will be a little shorter than usual. My apologies; next time I'll be more productive, I promise!

### Beginners' Track

We've done a lot of work with manipulating variables. However, sometimes you need to deal with tables of related data. Let's say you need variables containing the numbers 0 through 10, and to print the sum of these numbers:

```
value0 = 0
value1 = 1
value2 = 2
value3 = 3
value4 = 4
```

```
value5 = 5
value6 = 6
value7 = 7
value8 = 8
value9 = 9
value10 = 0
```

```
print value0 + value1 + value2 + value3 + value4 +
value5 + value6 + value7 + value8 + value9 + value10
```

The result, when you run this, is 55.

But this is a pretty annoying way to handle tables of data. You have to hardcode references to individual cells in your table, and the code is, frankly, silly.

This is where arrays come in. An array is a special kind of variable that has slots for multiple values. Imagine a row of cubbyholes, like this:

```
value: +-----+
       | | | | | | | | | |
       +-----+
```

---

# Programming: GSoft BASIC, part 3 of 6

Each slot in this array can contain one value. In this case, we want each slot to hold an integer. We create this array using the DIM statement, like this:

```
dim value(10) as integer
```

The DIM statement (short for DIMension) creates a one-dimensional array of integers, capable of holding 11 integer values. This array is named "value." Each cubbyhole is called a "cell."

You reference individual cells in the value array by using a subscript, which is a parenthetical expression, like this:

```
value(1) = 1
```

This puts the number 1 into the second cell in the array, so we see something like this:

```
+-----+
value: |  | 1 |  |  |  |  |  |  |  |  |
+-----+
```

Note that the first cell is cell number 0, the second is cell number 1, and so forth. If you want, you can ignore cell number 0, and just use cells 1-10, or whatever your array size is.

You can then read the number back from the array like this:

```
print value(1)
```

This prints out the second cell from the value array (the one at index 1).

What happens if you try to read a cell that hasn't been set? These values default to 0, but as a general rule it's considered poor programming practice to assume that a variable contains 0 unless you set it specifically.

Let's look at a version of our example program that uses arrays:

```
dim value(10) as integer

value(0) = 0
value(1) = 1
value(2) = 2
value(3) = 3
value(4) = 4
value(5) = 5
value(6) = 6
value(7) = 7
```

```
value(8) = 8
value(9) = 9
value(10) = 10
```

```
print value(0) + value(1) + value(2) + value(3) +
value(4) + value(5) + value(6) + value(7) + value(8) +
value(9) + value(10)
```

After this code runs, our array looks like this:

```
+-----+
value: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
+-----+
```

You can access these in any order you want to:

```
value(5) = 36
print value(10), value(5)
```

This will print:

```
10      36
```

But this isn't really much better than the first program: we're still setting up and printing the values individually. This is where a simple loop can be very handy. Let's take a look at the BASIC programmer's favorite loop, the FOR-NEXT loop:

```
dim value(10) as integer
dim index as integer
dim total as integer

for index = 0 to 10
    value(index) = index
next index

total = 0

for index = 0 to 10
    total = total + value(index)
next

print total
```

This version of our program uses two FOR-NEXT loops. Here's how these work:

```
FOR indexvariable = startvalue TO
endvalue
    <<<statements>>>
NEXT indexvariable
```

The FOR loop does a lot for us. It sets the index variable to the specified starting value, then runs the statements between the FOR and NEXT statements.

---

# Programming: GSoft BASIC, part 3 of 6

Then it loops back to the top, adds one to the index variable, and, if it's less than the end value, runs the statements continue, continuing to loop until the index variable is greater than the end value. This is equivalent to code like this:

```
indexvariable = startvalue
keepgoing:
    if indexvariable >= endvalue then
        goto exitloop
    end if
    <<<statements>>>
    goto keepgoing
exitloop:
```

Obviously using FOR-NEXT is a lot simpler.

The first loop in our sample program sets up the values in the array; the value at each index is sent to the value of the index variable. You can see how this works by converting this loop into the GOTO-equivalent code, as demonstrated above:

```
index = 0
keepgoing:
    if index >= 10 then
        goto exitloop
    end if
    value(index) = index
    goto keepgoing
exitloop:
```

The first time the loop executes, index is 0, so the statement "value(index) = index" means "value(0) = 0", and so forth. This is an incredibly useful tool.

The second loop simply adds up each entry in the array, adding the values to the total variable. Notice that it doesn't actually specify the index variable in its NEXT statement. That's because the index variable is optional here; in fact, providing it makes your program run a tiny bit slower—however, listing the index variable name in the NEXT statement makes GSoft BASIC better able to identify bugs in your program. Consider this code:

```
dim grid(4,4) as integer
dim x, y as integer

for x=0 to 4
    for y = 0 to 4
        grid(x,y) = x*y
    next x
next y
```

```
for x=0 to 4
    for y = 0 to 4
        print grid(x,y),;
    next y
print
next x
```

If you run this, you get the error "NEXT without FOR". That's because, if you look carefully, the "next x" and "next y" are in the wrong order in the first loop. The last FOR in a nested group of FOR statements must have its NEXT come first. In this example, it doesn't make much difference, but sometimes it will.

If you switch the two NEXT statements, everything will work as expected, printing a 5x5 grid of integers.

Arrays can have a subscript up to 32,767. Each array can be up to 65,536 bytes (64KB). Your array can have as many dimensions as you want; these are all valid arrays:

```
dim a(5) as integer
dim b(5,5) as integer
dim c(5,5,5,5,5) as integer
```

What if we want to print out the sums of the even indices into our array? This code will do it:

```
for index = 0 to 10 step 2
    print "value(";index;") = ";
value(index)
next
```

The STEP modifier to the FOR statement gives you great control; it lets you specify what to add to the index variable for each pass through the loop. You can use any value you want, either positive or negative. So you could even count backward, like this:

```
for index = 10 to 0 step -2
    print "value(";index;") = ";
value(index)
next
```

The second most popular type of loop is the WHILE-WEND loop. The WHILE-WEND loop looks like this:

```
WHILE condition
    <<<statements>>>
WEND
```

As long as the specified condition is true, the statements are executed. This can be converted into a GOTO loop like this:

---

# Programming: GSoft BASIC, part 3 of 6

```
top:
    if condition then
        goto finished
    end if
    <<<statements>>>
    goto top
finished:
```

The following sample program loops, waiting until the user hits a key:

```
while peek($00C000) < 128
    print "Waiting for a key!"
wend
poke $00C010,0
print "Thanks for the key!"
end
```

In this example, the condition is "peek(\$00C000) < 128". The peek function returns the byte value read from the specified memory location, and the POKE statement, which stuffs the specified byte value into a memory location. The address \$00C000 is the keyboard register; when it's greater than or equal to 128, a key has been pressed. \$00C010 is the keyboard strobe register; writing a value to this register clears the most recent key from the keyboard buffer.

There's one interesting quirk to this program, though—if the user hits a key before the while statement runs for the first time, they'll see "Thanks for the key!" printed without ever seeing the "Waiting for a key!" message. Depending on your program's needs, this could be a problem.

Here's one way to avoid this situation:

```
do
    print "Waiting for a key!"
loop until peek($00C000) >= 128
poke $00C010,0
print "Thanks for the key!"
end
```

This uses the DO-LOOP UNTIL mechanism. This loop always executes the code inside the loop at least once, and can be written using GOTO like this:

```
top:
    print "Waiting for a key!"
    if peek($00C000) >= 128 then
        goto finished
    else
        goto top
    end if
finished:
```

```
end if
finished:
```

The DO-LOOP mechanism is very flexible; there are several forms of it, and we won't be going into them all this time. If you're interested in learning more about it, look it up in the GSoft BASIC manual. The form we've seen here is the most commonly-used.

That's it for this issue. Next time we'll take a look at ways to embed data within your application, and some assorted commands that can be really useful.

## Advanced Track

This episode, we're going to continue our discussion of file I/O that we started last time, and we'll take a look at the use of pointers in GSoft BASIC; pointers will be critical when we begin talking about Toolbox programming in Part 5.

### ASSORTED FILE COMMANDS

There are a few handy file-related commands that don't really fit anywhere else in our discussion, which I'll touch on here. This isn't an exhaustive discussion, though; visit Chapter 14 of the GSoft BASIC manual, starting on page 203, for a more detailed discussion.

This example demonstrates the CURDIR\$ and DIR\$ functions:

```
dim filename as string
print "Directory of ";curdir$

filename = dir$("*")

while filename <> ""
    print " ";filename
    filename = dir$
wend
```

CURDIR\$ returns the current directory's pathname. DIR\$ returns a filename from the current directory. The argument is a filename. If you specify "\*", the name of the first file in the directory is returned. In this case, you can omit the argument for future calls; this will return the names of successive files in the directory. When the end of the directory is reached, an empty string is returned. If you specify an actual filename, either that file's name is returned (if the file exists) or an empty string is returned (if the file doesn't exist).

If you want to delete a file, you can use the KILL command:

---

# Programming: GSoft BASIC, part 3 of 6

```
kill filename
```

You can create a directory using the MKDIR command:

```
mkdir pathname
```

The NAME command renames files:

```
name filename as newfilename
```

For example, to rename the file "foo" to "FooBar", you can do this:

```
name Foo as FooBar
```

We'll come back to file I/O one more time, next time, to discuss random access files. These are special files in which records of a known, fixed size are written, which makes locating specific information very quick and easy.

## POINTERS

If you plan to do Toolbox programming (we're going to touch on this somewhat in Part 5), you'll need to use pointers and handles. A pointer indicates the memory address of a byte in memory, and a handle is an indirect pointer—a pointer to a pointer to a byte in memory. Handles are used by the Apple IIGS Memory Manager to track blocks of memory as they're moved around in your computer's RAM (we'll look at this in greater detail when we actually start using them).

You can create a pointer variable—a variable that can contain a pointer to another object—like this:

```
dim p as pointer to String
```

This creates a new variable, p, that can contain a pointer to a string. p has no value yet, though. You can assign the pointer like this:

```
dim p as pointer to String
dim s as String
s = "This is a string."
p = @s
```

Once these four lines of code have executed, p points to the string variable s, which contains the text "This is a string." s can then be passed to Toolbox calls that require pointers to C-strings (GSoft BASIC strings are null-terminated strings, called C-strings; they end with a byte with the value 0).

To access the data pointed to by a pointer variable, you need to \*dereference\* it. This is done using the dereference operator, ^.

You can print the string pointed to by p in the above example by using the following statement:

```
print p^
```

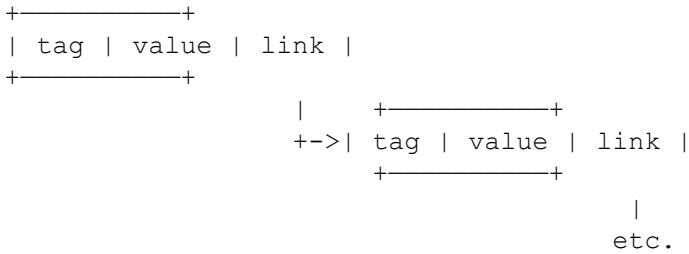
This will print the text "This is a string."

GSoft BASIC also lets you allocate blocks of memory that you can then reference, by pointer, to fill with data however you wish. This is done using the ALLOCATE function, like this:

```
type infoRec
    tag as string
    value as integer
    link as pointer to infoRec
end type
dim numbers as pointer to infoRec
dim p as pointer to infoRec
dim i as integer
numbers = NIL
for i=1 to 10
    allocate(p)
    p^.value = i
    p^.tag = "Text: "+str$(i)+". "
    if numbers = NIL then
        p^.link = NIL
        numbers = p
    else
        p^.link = numbers
        numbers = p
    end if
next
p = numbers
while p <> NIL
    print p^.value;" - ";p^.tag
    p = p^.link
wend
while numbers <> NIL
    p = numbers^.link
    dispose(numbers)
    numbers = p
wend
```

This example program implements a data structure called a linked list, in which a chain of records is built. Notice in the declaration of the infoRec type, how it contains a pointer to an infoRec, called link. This pointer lets each record point to the next one in the list, like this:

## Programming: GSoft BASIC, part 3 of 6



Each item in the list contains a string, called tag, and an integer value. The list is known to end when the link pointer is NIL. NIL is the name given to a special pointer value (which happens to be 0). We use this value to indicate that there aren't any more records in the list.

The fields in a record referenced by pointer are used using the `^` operator. To read the value field, you might do `numbers^.value`, for example.

Let's look over the code that demonstrates the use of the linked list. There are three variables created. The first, `numbers`, is a pointer to an `infoRec`. This is the head pointer for our linked list ("head pointer" is the term used to describe the main pointer to the first item in the list). `p` is a temporary pointer that we'll use while working with the list. `i`, an integer, is used as an index variable in our for loop. The head pointer is set to `NIL`, so we know there aren't any items in the list yet.

There are three loops in the program, each of which demonstrates a different concept. The first loop builds the linked list, the second walks the link list, printing out the contents of each item in the list, and the third loop deletes the list from memory.

The loop that builds the list is going to insert 10 items into the list. For each value, 1 to 10, it starts with `allocate(p)`. This allocates a new block of memory of the right size to contain a pointer to an `infoRec` record, then stores a pointer to the new memory block in `p`. The `ALLOCATE` function knows the size of the memory block to allocate based on the type of pointer; since `p` is a pointer to an `infoRec`, it knows to create a memory block the same size as an `infoRec` record.

Once the record has been allocated, the value is set to the index value, and the tag is set to a string that contains the index value. For the number 1, this will be "Text: 1."

Next, the new record needs to be inserted into the list. If the head pointer, numbers, is NIL, we know this is the first item to be added (obviously, in this case, we

know that item #1 is the first item, but by checking for NIL, our code is easier to reuse later, in cases where we don't always have advance knowledge of the contents of the records, such as when they're being read from a disk file). In this case, the link pointer in the p record is set to NIL (which indicates that there aren't any more records after the one we're adding), and the head pointer is set to point to the new record, p.

If the list isn't empty, we set the link pointer in the new record to point to the first item in the list (numbers), and then make the new record the first one in the list by setting numbers to point to it. This has the effect of making the new record the first one in the list; we're adding new records to the beginning of the list, instead of to the end. You can of course make your code so that you add each item to the end of the list, but the code is a little simpler this way, and data structures aren't a focus of this series.

Try doing this yourself on paper, following the code and sketching out the linked list. Trust me, doing this once or twice makes this a lot clearer.

The second loop prints out the values of each item in the list by walking the list. `p` is set to point to the first item in the list, then a while loop is used to scan through the list, printing the contents of the item, then setting `p` to the next item in the list. Once `p` becomes `NIL`, the end of the list has been reached and the loop exits.

The last loop disposes of the list by walking it, deleting each item in the list. It sets `p` to the link pointer of the list's first record, then disposes of the first record. By doing this, we can keep a pointer to the new first item in the list. `numbers` is then set to `p`. The result of this is the removal of the first item in the list. This continues until the list is empty, at which point `numbers` is `NIL` and the loop exits.

The `DISPOSE` function deletes a block of allocated memory, making the memory available for future use. You should always remember to delete memory you've allocated; if you forget, memory will eventually fill up, and things will start to fail.

That's it for this time. Next time we're going to finish our discussion of files by talking about random-access files, and we'll have a look at error handling in GSoft BASIC. We'll start putting together a little program that combines our work on pointers and records with the file I/O stuff we've been covering, to create a simple database program.



# Bring those skills to a higher level

Welcome back! This issue, in the beginners' track, we're going to look at ways to embed data right into your program, along with a few other handy commands that we've missed along the way.

In the advanced track, we'll wrap up our look at disk files by covering binary and random-access files, and we'll look at adding error handling to your program.

## Beginners' Track

Sometimes your program needs access to some constant data. For example, a program that computes tax information needs access to a tax table, in which you can look up, based on a person's taxable income, the amount of tax they owe.

Sometimes you can keep this data in a separate file. There are times when this is the best way; if the data might need to be periodically revised (either by the user or by you, the programmer), it might be better to keep the data in a separate file that you read in using the file I/O commands we've been covering. On the other hand, if the data is unchanging, such as characteristics of chemical compounds, you might want to embed it directly into the program.

GSoft BASIC provides three standard BASIC statements used for embedding and accessing data within your program. These are DATA, READ, and RESTORE.

The DATA statement lets you embed data within the program. The data can be of any type, string or numeric, and you can mix and match:

```
DATA Apple, 100, "Apple IIgs", 5.5
```

This creates four data items: the text APPLE, the number 100, the text "Apple IIgs", and the number 5.5. Note that without quotes, strings are converted into all upper-case. You can also use hexadecimal numbers:

```
DATA $20, $ED, $FD, $60
```

Each data item is scanned using the same rules as the INPUT statement. You can put as many data items on each DATA statement as you like.

Once you've established your data, you can read it using the READ statement, like this:

```
dim s as string
dim percent as integer
dim apple as string
dim num as double

READ s, percent, apple, num
```

This reads in a string, then an integer, then another string, then a number. Reading begins with the first data item in your program and continues onward through them. They don't have to match up in terms of the number of items per line. So you could just as easily have:

```
DATA 5, 10, 15
DATA 20
DATA 25, 30

READ x, y
READ z, a, b
READ c
```

Note that if you READ more items than have been created using DATA, an error will occur. We'll have a sample program that makes use of DATA and READ later in this article.

## GRAPHICS, PART 1

There are two ways to do graphics in GSoft BASIC: you can use the integrated GSoft BASIC graphics commands (there are only three of them) to do simple graphics, or you can use the QuickDraw II Tool Set to do more advanced graphics. We'll look at some basic QuickDraw functions late in the Advanced Track, but let's start with the GSoft BASIC graphics commands in this program:

```
dim color as integer
dim x as integer
dim key as string

hgr
for color = 0 to 15
    hcolor= color
    for x = color * 20 to color * 20 + 20
        hplot x,0 to x,200
    next
next
get key
text
end
```

---

# Programming: GSoft BASIC, part 4 of 6

This program draws a test pattern of bars on the screen. Go ahead and give it a try.

Here's how it works. The HGR command puts the IIGS into the 320x200 super-hires graphics mode and clears the screen to black. In this mode, there are 16 colors available, numbered 0 to 15. Later we'll learn how to specify which colors to use (you can pick which out of a total of 4,096 colors to use), but for now, look on page 208 of the GSoft BASIC manual for a list.

We enter a loop to loop over all 16 colors, each time incrementing the variable color to match the next color, starting at color 0 and making our way toward color 15. The HCOLOR= command is used to set the current drawing color to the one indicated by the color variable. Note that the command is "HCOLOR="; putting a space between HCOLOR and the "=" will make this not work.

A second loop is used to pick the X-coordinate to draw at. We'll draw 16 bars that are each 20 pixels wide ( $16 \times 20 = 320$ ) across the screen, each in a different color. Let's look at how the math in this for loop works.

If the current color is color 0, the math for the start and end values of the loop are:

```
color * 20 = 0 * 20 = 0
color * 20 + 20 = 0 * 20 + 20 = 20
```

This gives us a 20-pixel range. For color 15, the math works out like this:

```
color * 20 = 15 * 20 = 300
color * 20 + 20 = 15 * 20 + 20 = 320
```

This is exactly what we want.

Next comes the HPLOT command, which actually draws a line on the screen. In this case, we're drawing a line from (x,0) to (x,200). On the computer screen, x (the horizontal position) runs from left to right, and y (the vertical position) runs from top to bottom. This is different from in algebra and trigonometry, where y runs from bottom to top.

The loops end after this, and we wrap up, after the entire screen is drawn, we wait for a keypress using the GET statement, which waits for a single key to be pressed and stuffs that key into the indicated variable. The TEXT command switches back to the text screen.

The HPLOT command is the most interesting of the GSoft BASIC graphics commands, and it has several forms:

```
hplot x,y : ! Draws a point at (x,y)
hplot x1,y1 to x2,y2 : ! Draws a line
from (x1,y1) to (x2,y2)
hplot x1,y1 to x2,y2 to x3,y3 to
x4,y4 : ! Draws a line chaining these
points end-to-end
hplot to x,y : ! Draws a line from the
last point drawn to (x,y)
```

These provide the basic tools needed to create simple graphics.

## A COMBINED EXAMPLE

Let's take a look at a program that uses DATA statements to define the points to draw.

```
dim color as integer
dim x,y as integer
dim key as string
dim done as boolean
dim newLine as boolean

done = false
newLine = true
hgr
do
    read color,x,y : ! Read the color and
position
    select case color
        case -1:
            if x = -1 then
                done = true
            else
                newLine = true
            end if
        case else:
            hcolor= color
            if newLine = true then
                hplot x,y
                newLine = false
            else
                hplot to x,y
            end if
        end select
    loop until done = true

get key
text
end

! Frame
```

---

# Programming: GSoft BASIC, part 4 of 6

```
data 7,0,0, 7,0,199, 7,319,199, 7,319,0,
7,0,0
data -1,0,0
data 5,20,20, 5,20,180, 5,300,180,
5,300,20, 5,20,20
data -1,0,0

! "H"

data 9,80,50, 9,95,50, 9,95,95,
9,120,95, 9,120,50
data 9,135,50, 9,135,150, 9,120,150,
9,120,110
data 9,95,110, 9,95,150, 9,80,150,
9,80,50
data -1,0,0

! "I"

data 9,145,50, 9,200,50, 9,200,65,
9,180,65
data 9,180,135, 9,200,135, 9,200,150
data 9,145,150, 9,145,135, 9,165,135
data 9,165,65, 9,145,65, 9,145,50
data -1,0,0

! "!"

data 9,210,50, 9,225,50, 9,225,130,
9,210,130, 9,210,50
data -1,0,0
data 9,210,135, 9,225,135, 9,225,150,
9,210,150, 9,210,135
data -1,0,0

! End of graphic

data -1,-1,-1
```

Key this program in and run it. It draws a frame around the screen in red and orange, then prints in big block letters a greeting in the middle of the screen, in yellow. We use a big loop to read in the data variables (using the READ statement) one at a time and process them.

Each data point consists of three values. The first is a color; the second is the X coordinate of the point to draw, and the third is the Y coordinate of the point. If the color is -1, this indicates a special command. The command is specified using the X coordinate value. If this is 0, the currently line should end and a new one should begin (the next point will start a new line). If X

is -1, drawing is complete and the loop should terminate.

The loop reads in the color and X and Y coordinates into variables called color, x and y. A select-case statement is used to check the color to see if it's a special command. If it's -1, and x is also -1, done is set to true; this will terminate the loop. If x is 0, the newLine flag is set to true; this will be used when drawing the point to decide if HPLOT x,y should be used, or HPLOT TO x,y.

If the color isn't -1, we use HCOLOR= to set the drawing color; then we plot. If newLine is true, we use HPLOT, otherwise HPLOT TO.

Once drawing is complete, a GET command is used to wait for the user to press a key, then text mode is restored and the program ends.

The DATA statements define the frame and the letters that are drawn by this program. You can experiment with the DATA statements to create your own graphics. The line:

```
DATA -1,0,0
```

indicates that the current line segment (such as the letter "H", which is drawn as one line starting at the top left corner and running clockwise) is finished and that the next data point starts a new object.

The line:

```
DATA -1,-1,-1
```

indicates that drawing is finished.

This program shows how READ and DATA can be used in a real program, and is a good example of graphics in GSoft BASIC as well.

This is the end of the Beginner's Track. The remaining articles in this series will be entirely devoted to the Advanced Track. We haven't covered every possible GSoft BASIC topic that beginners will want to know about, but we've had a good look at the most important issues. At this point, if you've been following the Beginner's Track, you should go back to the first article (Volume 4, Issue 1) and start the Advanced Track.

## Advanced Track

### BINARY FILES

GSoft BASIC has a number of commands that make it possible to read and write binary files efficiently.

---

# Programming: GSoft BASIC, part 4 of 6

This program writes four bytes of data into a file:

```
dim a(3) as byte
dim i as integer

open "filename" for binary as #1
a(0) = 10
a(1) = 20
a(2) = 40
a(3) = 80
for i = 0 to 3
    put #1, , a(i)
next
close #1
end
```

We specify "binary" as the access type when opening the file; this lets us read and write binary data to the specified file.

The key is the PUT command used here. PUT writes the results of an expression to the file:

```
PUT #<filenumber>, <fileposition>,
<expression>
```

This writes into the file given by the filenumber, at the byte offset indicated by fileposition, the result of the given expression. If you want to just write at the current position (which we did in the example above), just leave out the file position argument (but NOT the corresponding commas!). This looks a little funny if you're used to Applesoft BASIC, but Microsoft BASIC programmers have seen this kind of thing before.

You can read this file (and view the contents) using this code:

```
dim a as byte
open "filename" for binary as #1
while not eof(1)
    get #1, , a
    print a
wend
close #1
end
```

The EOF function returns true if the end of the file has been reached, otherwise it returns false. The GET statement retrieves a value from the disk file, storing it in the appropriate variable (in this case, a).

You don't have to use simple variables, either. If you want to write a record into a file, you can do that. And that's a great thing to be able to do when you're using random access files.

## RANDOM ACCESS FILE I/O

Last time, we talked about basic sequential files. A sequential file is one in which data is written in more or less a straight shot, from the beginning of the file to the end. This time, we're going to talk about random access files. Random access files are files in which records are written at known intervals throughout the file, so you can read back in specific records easily at a later time. Databases use random access files, for example.

GSoft BASIC provides special features that make it easier to work with random-access files. Let's take a look at these. This sample program creates a very simple phone list database, which you can add entries to and search based on people's last name. We'll skim over most of the program, it's mostly simple video display and record management, which we've done before.

```
type personRecord
    firstname as string : ! up to 15
    characters
    lastname as string : ! up to 20
    characters
    phone as string : ! up to 20
    characters
    id as integer : ! user-defined
ID
end type

const personsize = 80 : ! reserve 80
bytes per record

dim done as boolean

! Main loop, MainMenu returns true
! when program should quit.

done = false
while not done
    done = MainMenu
wend
end

! Handle the main menu

function MainMenu as boolean
    dim ch as string
    dim result as boolean
    dim retry as boolean

    result = false
    home
    call PrintCentered("Phone List Demo
for Juiced.GS", (true))
```

---

# Programming: GSoft BASIC, part 4 of 6

```
    call HorizLine("=", 79)
    call DrawBox(20,8,60,17)
    vtab 9
    call PrintCentered("Main Menu",
(true))
    vtab 10 : htab 21 : call
HorizLine("-", 39)
    vtab 11 : htab 22 : print "1. Add a
person"
    vtab 13 : htab 22 : print "2. Find a
person"
    vtab 15 : htab 22 : print "3. Quit"
    vtab 23 : htab 1 : call
HorizLine("=", 79)
    vtab 24 : htab 1 : print "Choose an
option (1-3): ";

    ! Keep trying until a valid choice
is made.

do
    retry = false
    get ch
    vtab 24 : htab 25 : print ch;

    select case ch
        case "1": call AddPerson
        case "2": call FindPerson
        case "3": result = true
        case else:
            vtab 24 : htab 25 : ? "
";chr$(7); : htab 25
            retry = true
        end select
    loop until retry = false

    MainMenu = result
end function

sub AddPerson
    shared personsize
    dim person as personRecord
    dim recnum as integer
    call DrawBox(50,10,76,22)
    call EraseBox(51,11,75,21)
    vtab 11 : htab 57 : ? "Add a Person"
    vtab 12 : htab 51 : call
HorizLine("=", 25)
    vtab 14 : htab 52 : input "First:
";person.firstname
    vtab 16 : htab 52 : input " Last:
";person.lastname
    vtab 18 : htab 52 : input "Phone:
";person.phone
```

```
    vtab 20 : htab 52 : input "    ID:
";person.id

    ! Add to the end of the file.

    open "phonelist" for random as #1
len personsize
    recnum = lof(1) + 1
    put #1,recnum,person
    close #1
end sub

sub FindPerson
    shared personsize
    dim i as integer
    dim count as integer
    dim who as string
    dim person as personRecord
    dim ch as string

    call DrawBox(5,5,55,11)
    call EraseBox(6,6,55,11)
    vtab 6 : htab 23 : print "Find a
Person"
    vtab 7 : htab 6 : call
HorizLine("=",49)
    vtab 9 : htab 6 : input "Last name:
";who

    ! Start searching the file

    open "phonelist" for random as #1
len personsize
    count = lof(1)
    for i=1 to count
        vtab 10 : htab 6 : print
"Searching... ";
        get #1,i,person
        if person.lastname = who then
            call EraseBox(6,8,55,11)
            vtab 8 : htab 6 : print
"Name: ";person.lastname;";
";person.firstname
            vtab 9 : htab 6 : print
"Phone: ";person.phone
            vtab 9 : htab 40 : print
"ID: ";person.id
            vtab 10 : htab 6 : print
"Press a key to continue: ";
            get ch
            end if
        next
    close #1
```

---

# Programming: GSoft BASIC, part 4 of 6

```
end sub

sub PrintCentered(s as string, newline
as boolean)
    dim c as integer
    c = (80-len(s))/2
    htab c
    print s;
    if newline then print
end sub

sub HorizLine(c as string, l as integer)
    dim i as integer
    for i=1 to l
        print c;
    next
end sub

sub EraseBox(left as integer, top as
integer, right as integer, bottom as
integer)
    dim width, height as integer
    dim i,j as integer

    width = right - left
    height = bottom - top
    for i=1 to height
        vtab top + i - 1
        htab left
        for j=1 to width
            print " ";
        next
    next
end sub

sub DrawBox(left as integer, top as
integer, right as integer, bottom as
integer)
    dim count as integer
    dim i as integer

    vtab top
    htab left
    print "+";
    count = right-left-1
    for i=1 to count
        print "-";
    next
    print "+"
    vtab bottom
    htab left
    print "+";
    for i=1 to count
        print "-";
```

```
next
print "+";

! Do the sides

count = bottom-top-1
for i=1 to count
    vtab top+i
    htab left
    print "|";
    htab right
    print "|";
next
end sub
```

The AddPerson function adds one new record to the end of the file. It displays a box on the text screen and asks for the person's first name, last name, phone number, and an ID number (just to demonstrate that you can use integers, or any type, in your random-access records).

Once this information has been input, we open the file "phonelist" for random-access with the length specified as personsize, which is the size of one record. Then we figure out what record number to write. We want to add to the end of the file, so we use the LOF (length-of-file) function to find out how many records are in the file, then add one to get the new record number.

Then we use the PUT statement to write the new record to the file, and close the file.

FindPerson begins by asking the user for the last name of the person to find. It then opens the "phonelist" file, finds the number of the records in the file, and begins a loop to look at every record in the file.

For each record, we use the GET statement to read the record into the person record in memory, and we look to see if person.lastname matches the name the user entered. If it does, we clear out the little text box and display the contents of the record, then wait for a keypress and continue searching the file.

Once all records have been searched, we close the file and return.

This is a very simple database. You might try adding functions to delete records, edit records, and perform more substantial searches. You might also want to add some error handling, as discussed in the next section, to make the program more reliable. Something that would be very good to add would be code to make sure that the lengths of the fields (the strings entered by the user) don't make the record longer than 80 bytes. Keep in



---

# Programming: GSoft BASIC, part 4 of 6

mind that each string requires 5 bytes plus the length of the string itself.

Notice the use of VTAB (vertical tab) and HTAB (horizontal tab) to create more dynamic text displays, and the use of the DrawBox, HorizLine, and EraseBox functions to create a rudimentary window system on the text screen.

## ERROR HANDLING

Up until this point, when an error has occurred in one of your programs, it's quit running and printed some sort of error message to the screen. This is fine while you're debugging, but when you want to present a real application to end users, this isn't acceptable at all.

To handle errors, you need to install a error handler. This is done using the ONERR GOTO statement. Let's look at a simple sample program:

```
dim i as integer
input "Enter an integer: "; i
print "Here's your integer: ";i
end
```

This program simply inputs an integer from the user, then prints it back out. If you enter a value that's too large to be an integer (such as 392834), you get an integer overflow error, like this:

```
#run error.bas
Enter an integer: 392834
Integer overflow
>>>      INPUT "Enter an integer: ";i
```

Unacceptable for a final program. Let's use ONERR GOTO to correct for this:

```
dim i as integer
dim answer as string

onerr goto errorHandler
input "Enter an integer: "; i
print "Here's your integer: ";i
end

errorHandler:
  select case err
    case 19:
      print chr$(7);"Integers can
only be from -32768 to 32768. Try again."
      resume
    case else:
      print chr$(7);"An unexpected
error occured (#";err;"). Try again (Y/N)?
";
```

```
get answer
print answer
if answer = "n" or answer = "N"
then end
      resume
end select
end
```

This version installs, using ONERR GOTO, an error handler that begins at the label errorHandler. When an error occurs, GSoft BASIC will GOTO this label instead of quitting and displaying an error message.

The errorHandler code looks at the function, ERR, to determine which error occurred. Error number 19 is the integer overflow error. In this case we print a description of the problem and ask the user to try again. The RESUME statement returns program control to the beginning of the same line of code that caused the error in the first place. Note that if you have multiple commands on a line, as in the following example, if the error occurs in any command on that line, the entire line still gets run again.

```
print "Please enter an integer: "; :
input i
```

The prompt here will be displayed again after the RESUME.

If any other error occurs, we print a message indicating that an unexpected error occurred, including the error number. We then ask the user whether they want to retry. If they press "n" or "N" we end the program, otherwise we RESUME.

If you want to cancel the effects of an ONERR GOTO (to remove error handling), use the line:

```
ONERR GOTO 0
```

You can change error handlers multiple times within the same program by simply using a new ONERR GOTO statement. If your program uses line numbers, the ERL function will return the line number on which the error occurred:

```
errHandler
  print "Error #";err;" occurred on
line ";erl;".
end
```

If the line on which the error occurred doesn't have a line number, ERL returns 0.

Next issue, we'll begin looking at using the Apple IIGS Toolbox in your programs.

## Using the Toolbox

Starting with this issue, we're all about the advanced track! This month we're going to take a look at basic Toolbox programming concepts, including loading and unloading tool sets, starting up and shutting down tool sets, and basic QuickDraw graphics programming.

If you plan to do any significant Toolbox programming, you should get the *Toolbox Reference* books from The Byte Works. There are three volumes, plus an additional volume, the Programmer's Reference for System 6.0.1, which includes additional information about new functions provided by Systems 6.0 and 6.0.1. It's not my job to provide information on every Toolbox call that exists; there are hundreds of them!

### TOOLBOX ORGANIZATION

The Toolbox is a collection of tool sets. Each tool set is made up of a number of toolbox calls that serve related purposes. For example, the QuickDraw tool set contains graphics drawing functions, the Window Manager contains functions for creating and managing windows, and so forth.

Each tool set has six special "housekeeping" functions that are used to start up, get information about, and shut down the tool set. These functions are named by taking a name representative of the tool set (such as "QD" for QuickDraw) and appending the name of the function afterward:

`xxxBootInit` (such as `QDBootInit`) is called by the operating system when the tool set is loaded from disk (or when the computer is booted, if the tool set is in ROM). You'll never call this function yourself—it's not allowed.

`xxxStartUp` (such as `QDStartUp`) is the function your application calls to start up the tool set so you can use it. All tool sets must be started up before you can use them.

`xxxShutDown` (such as `QDShutDown`) shuts down the tool set; your application should call this before it quits.

`xxxReset` (such as `QDReset`) is called by the operating system when the computer is reset. You're not allowed to call it.

`xxxStatus` (such as `QDStatus`) returns boolean true if the tool set is started up, otherwise it returns false.

`xxxVersion` (such as `QDVersion`) returns the version number of the tool set. This 16-bit value, as a hexadecimal number, is formatted like this: `$xMmb`, where M is the major version number, m is the minor version number, and b is the bug fix version number. So version 1.0 would be `$0100`, and version 2.1.3 would be `$0213`. The high nibble, x, is used as special flags by a couple of tool sets, so you should ignore them.

### ABOUT RESOURCES

Resources are data objects attached to a file by storing them in the file's "resource fork." They can then be easily fetched and used by software using calls to the Resource Manager tool set, or by using other Toolbox functions that support resources.

Many toolbox functions accept as input references to data structures. For example, when you want to create a new window, you pass a reference to a window parameter list structure to the `NewWindow` or `NewWindow2` function. In good, modern Apple IIGS applications, this reference will be via a resource.

Because the proper way to do things nowadays is via resources, that's how we're going to do it. The trick is that if you don't have software to create the resources, you're going to have a rough time of it.

There are a few solutions. I'm going to use the Rez resource compiler, developed by Apple. Creating resources in Rez is done by entering text "code" describing the layout and contents of the resources, which is then compiled into Rez code. Rez must be used from within the ORCA or APW shell, and is included with every ORCA or APW language. Please note that it's beyond the scope of this series to cover Rez programming, but if you have Rez, you should have documentation for it as well; every ORCA manual has a very good chapter on Rez.

You can also create resources using a resource editor. Genesys, a commercial product (no longer available) will work for our purposes. Foundation, a freeware resource editor, doesn't really have all the features you'll need, even for the simple programs we'll be doing.

If you already have a resource editor you like, go ahead and use it! If not, you can get this month's source code, the Rez source, and compiled resources on disk from the *Juiced.GS* Collection Shareware '99/Fall.

---

# Programming: GSoft BASIC, part 5 of 6

## BASIC TOOL SET THEORY

Some tool sets are always available. Other tool sets are kept on disk, and need to be loaded into memory when your application starts running, and unloaded when your application quits.

Tool sets that are kept in ROM on both ROM 01 and ROM 3 computers include the Tool Locator, Memory Manager, Miscellaneous Tool Set, QuickDraw, Event Manager, and Desk Manager. There are a few others, but you're not likely to use them in your early programs.

If a tool set is guaranteed to be available, such as these are, you can just start it up and use it. If you don't know for certain that the toolset will be in memory already, your application will have to check to see if it's there, and load it if it's not. There is a handy Tool Locator function, `StartUpTools`, that will do this for you, and we'll have a look at it shortly, but if you're going to properly understand the Toolbox, you'll need to know how to do this stuff by hand.

To load a tool set from disk, you need to call the `LoadOneTool` function in the Tool Locator before you start it up. Then, before you quit (and after you shut the tool set down), you call `UnloadOneTool` to shut it down. You'll need to know the tool set number for the tool set you want to load.

```
! Load the QuickDraw Auxiliary Tool Set

LoadOneTool($12, $100)
if (toolError <> 0) then
    ! handle the error, couldn't load
the tool set
end if

QDAuxStartUp
...
WaitCursor
...
QDAuxShutDown
UnloadOneTool($12)
```

This example loads the QuickDraw Auxiliary Tool Set, tool set number \$12 (18 decimal). The `LoadOneTool` call takes as arguments the tool set number and the minimum version number you're willing to accept. In our case, we don't care what version it is, so we specify \$0100, version 1.0, which means that any version 1.0 or later will be accepted.

If an error occurs in `LoadOneTool` (such as error \$0001, tool set not found, or error \$0110, specified

minimum version not found), we have to handle that, since the tool set wasn't loaded.

If the tool set was loaded successfully, we can call `QDAuxStartUp` to start the tool set up. Then we can call functions in the tool set, such as `WaitCursor`, which displays the wristwatch "busy" cursor (FYI, the `QuickDraw` function `InitCursor` restores the arrow cursor).

When our application is about to exit, we call `QDAuxShutDown`, then unload it by calling `UnloadOneTool`.

A couple of caveats to this example: `QuickDraw` Auxiliary will only work if `QuickDraw` is already running, so you have to start `QuickDraw` first. Also, the mouse cursor is only available if you've also started up the Event Manager.

Some tool sets' `xxxStartUp` functions have input arguments, and many of them require some direct page memory be allocated for their use, and a pointer passed into the function (direct page memory, briefly, is memory located in bank \$00 of memory, which can be accessed more quickly by software for higher-performance storage of working data).

As you can guess, if your application uses a half-dozen or more tool sets, this can get pretty tedious. Even with the `LoadTools` function in the Tool Locator, which loads multiple tools at once, things still drag on, with all the starting up and shutting down of tool sets. Fortunately, there's an easier way.

## THE EASIER WAY

After years of programmers complaining about all that work starting up and shutting down tool sets, Apple added the `StartUpTools` and `ShutDownTools` in System 5.0. The `StartUpTools` function, given a list of tool sets and minimum version numbers, automatically loads the tool sets, allocates direct page space and calls the `xxxStartUp` functions for each of the listed tool sets. Likewise, `ShutDownTools` calls the `xxxShutDown` functions, deallocates direct page space, and unloads the tool sets.

And there was much rejoicing. And when the rejoicing stopped, the programmers rejoiced again. Then they partied. Life was good. Really, really good.

## A BASIC SHELL

Let's have a look at an incredibly trivial shell program that starts up some standard tool sets, waits

---

# Programming: GSoft BASIC, part 5 of 6

for the mouse button to be clicked, then shuts down the tools and quits. Remember that this is in two parts, the BASIC source code, then the Rez code for the resources. If you get the *Juiced.GS* disk containing these files, these are Tools1.bas and Tools1.rez.

```
!
! Toolbox startup and shutdown
! using StartUpTools and ShutDownTools.
!

dim userID as integer
dim startStopParam as long

! Set up

if InitToolbox <> 0 then
    call StopToolbox
    print "Unable to start up the tools
properly."
end
end if

! Main program

InitCursor
while Button(0) = 0
wend

! Shut down

call StopToolbox          : ! Shut
down the tools
end

function InitToolbox as integer
    shared userID
    shared startStopParam

    TLStartUp
    userID = MMStartUp
    startStopParam =
StartUpTools(userID, refIsResource, 1)
    InitToolbox = toolError
end function

sub StopToolbox
    shared userID
    shared startStopParam

    ShutDownTools(refIsHandle,
startStopParam)
    MMShutDown(userID)
    TLShutDown
end sub
```

The main program starts by calling our InitToolbox function to set up the tools. InitToolbox works by calling TLStartUp (to start the Tool Locator, which handles loading other tool sets), then by calling MMStartUp to start the Memory Manager. MMStartUp returns a user ID, which identifies the application. Finally, StartUpTools is called to load the needed tool sets and start them up, all in one swift action.

The parameters for StartUpTools are our user ID, a "reference descriptor", which tells the function what type of reference the last value is, and a reference. The reference can be a pointer (which we've talked about before), a handle (a pointer to a pointer, but there's a formal definition involved which we'll cover later), and the resource ID of an rToolStartup resource that lists the tool sets we want to start up.

StartUpTools returns as a result a reference to a new startStopRecord, which we'll pass to ShutDownTools. It includes information on which tool sets were successfully started, what our resource fork file ID number is (we won't worry about this), and other information, most of it stuff we don't need to know, but it's there for the ShutDownTools call to use. StartUpTools automatically opens our application's resource fork for us, giving us access to our resources.

InitToolbox returns the value of toolError. This is a special function that returns the error code reported by the last Toolbox call, in this case StartUpTools. Our main program looks at this value, and if it's not zero (zero means no error occurred), we shut down the tools by calling StopToolbox, print an error message, and quit the program.

The main program calls InitCursor to switch from the wristwatch cursor (StartUpTools sets the cursor to the watch cursor, indicating that the program is still loading) to the arrow cursor, then enters a while loop that calls the Event Manager function Button. Button returns a boolean value that's true if the specified mouse button is down, false if it's not. We simply wait until button 0 (the only button on a standard Apple IIGS mouse) is pressed, then the loop exits.

When the loop is finished, we call our StopToolbox function to shut down the tools. This is done by first calling ShutDownTools, passing the startStopRecord returned by StartUpTools, then by calling MMShutDown (passing in our user ID), and then, finally, TLShutDown. Note that we shut down tools in the opposite order that we started them up. This is very important, because tools tend to rely on each other. For example, the Menu Manager needs the Control

---

# Programming: GSoft BASIC, part 5 of 6

Manager, which needs QuickDraw, so you have to start QuickDraw first, then Control Manager, and finally the Menu Manager. StartUpTools handles this for us. We love StartUpTools.

This is an incredibly simple demonstration program, and we only have one resource, the tool startup table used by StartUpTools. Here's the Rez source:

```
#include "types.rez"

// The rToolStartup resource contains
information on the graphics mode
// we want and the tool sets we need.

resource rToolStartup(1) {
    mode320,
    {
        3, $0302,          /* Misc Tool */
        4, $0307,          /* QuickDraw II */
        5, $0304,          /* Desk Manager */
        6, $0301,          /* Event Manager */
        11, $0300, /* Integer Math Tool

Set */
        14, $0303,         /* Window Manager */
        15, $0303,         /* Menu Manager */
        16, $0303,         /* Control Manager */
        18, $0304,         /* QuickDraw II

Auxiliary */
        19, $0301,         /* Print Manager */
        20, $0303, /* LineEdit Tool Set */
        21, $0304,         /* Dialog Manager */
        22, $0301,         /* Scrap Manager */
        23, $0303,         /* SFO */
        27, $0303,         /* Font Manager */
        28, $0303,         /* List Manager */
        34, $0103 /* TextEdit Tool Set */
    }
};
```

The first field indicates the graphics mode your application will use. It's actually a master scan control byte (SCB), with some additional flags. The SCB indicates graphics mode (320 or 640), whether or not to enable fill mode, and the like. This isn't very important right now. The point in this case is that we've chosen 320 mode.

The remainder of the resource lists out the tool sets that we want started up, and the minimum version number of each tool set that we're willing to accept. This particular list includes just about everything the average application needs.

They don't actually have to be listed in any particular order—StartUpTools will sort them out—but it makes it easier to keep track of what tools you're starting up if you keep them in numerical order.

Note that we don't list the Resource Manager. That's always started up by StartUpTools, so we don't have to include it. And since we start up the Tool Locator and Memory Manager ourselves, we don't list them either.

Once you've entered the Rez code, you can compile it and add it to the BASIC program by typing the following command in the ORCA Shell:

```
compile Tools1.rez keep=Tools1.bas
```

The copy of Tools1.bas included on the available *Juiced.GS* disk has the resources already attached.

## GRAPHICS, PART TWO: BASIC QUICKDRAW

Let's add code to our example to open a window and draw some stuff into it. If you get the disk from *Juiced.GS*, these files are Tools2.bas and Tools2.rez. First, let's look at a new function, NewDocument, that creates a window:

```
function NewDocument as GrafPortPtr
    NewDocument = NewWindow2(NIL, 0,
    NIL, NIL, refIsResource, 1000, rWindParam1)
    if toolError <> 0 then
        print "Error opening window."
    end if
end function
```

This code uses the rWindParam1 resource with resource ID 1000 to create a new window. The first parameter to NewWindow2 is a pointer to a new title for the window. The window template actually specifies a title for us, and we want to use that default title, so we specify NIL, indicating that we don't want to replace the title.

The second argument is called the refCon, the "reference constant." This is a long integer that you can use for whatever you want. We don't need it, so we set it to zero. The third argument is a pointer to the subroutine in our program that will handle drawing the window's contents when the window has to be redrawn (for example, if another window is drawn in front of it, then moved). We'll look at how to do that next time.

The fourth argument is a pointer to the window definition procedure (usually called a "window defproc"). The window defproc lets you (in theory) create

---

# Programming: GSoft BASIC, part 5 of 6

custom window appearances. In reality it doesn't work very well. You should always specify NIL for this argument.

The fifth and sixth arguments are the window parameter table reference description and reference. In our case, we're specifying a resource, so we use `refIsResource`, and the reference is the resource ID of our window template resource.

The last parameter is the resource type used to define the window. This can be either `rWindParam1` or `rWindParam2`. However, `rWindParam2` is designed to be used when you're using a custom window defproc, so you'll almost always say `rWindParam1`.

When `NewWindow2` returns, you'll get back a pointer to the new window's `GrafPort` (a `GrafPortPtr`). A `GrafPort` is a conceptual device that represents a portal through which the user can look at a portion of a large document. It's not quite the same thing as a window.

The main difference between a `GrafPort` and a window is that a window has a frame, and might have a title bar, close box, grow box, zoom box, scroll bars, and so forth. A `GrafPort` has no visible representation onscreen, except that you can draw into it. It just happens that every window record contains a `GrafPort` defining where the window's drawing area is, so we usually access windows as if they were `GrafPorts`. We'll discuss `GrafPorts` more completely next time.

If an error occurs in `NewWindow2`, `toolError` will be nonzero. If it is, we need to handle that error. For our example, we're simply printing a message. In the real world we'll need to do something more useful, such as display an alert informing the user of the error. We'll talk about displaying alert boxes next time as well.

Once the window has been created by `NewWindow2`, we return that window pointer to the caller. This `NewDocument` function can in fact be called more than once, to create multiple windows, all identical. You can also change it to alter the name of the window, the position of the window, and so forth.

Our program needs two more global variables added at the top:

```
dim myWindow as GrafPortPtr
dim r as Rect
```

Let's look at the new main body of our program. This can be simply dropped into the previous example, replacing the code between the comments `"! Main program"` and `"! Shut down"` with the code below:

```
myWindow = NewDocument
SetPort(myWindow): ! Draw in the window

SetSolidPenPat(3): ! Set the pen color
MoveTo(10,10)
LineTo(200,10)
LineTo(200,80)
LineTo(10,80)
LineTo(10,10)

SetSolidPenPat(5)
r.left = 30
r.top = 50
r.right = 80
r.bottom = 90
FrameOval(r)

SetSolidPenPat(7)
r.left = 40
r.top = 60
r.right = 70
r.bottom = 80
PaintRect(r)

r.left = 50
r.top = 65
r.right = 60
r.bottom = 75
EraseRect(r)

InitCursor
while Button(0) = 0
wend
```

The first thing we do is create a new window by calling the `NewDocument` function. Then we can start drawing into it. We first call the `QuickDraw` function `SetPort` to select our new window as the current `GrafPort`. Then we call `SetSolidPenPat` to set the current pen color (the "solid pen pattern") to color 3 (there are sixteen colors in 320 mode, numbered 0 through 15. Zero is black and 15 is white, and of course you can change the colors if you want to).

Then we call `MoveTo` to move the drawing pen to the coordinates 10,10. This puts the pen 10 pixels across and 10 pixels down inside the current `GrafPort` (our window's port). Then with four `LineTo` calls, we draw four lines creating a rectangle, specifying each time an appropriate X,Y coordinate pair.

After we draw our lines, we call `SetSolidPenPat` to change the color again, and then we set up the rectangle record `r` (`Rect` is a standard `QuickDraw` data type) to



---

# Programming: GSoft BASIC, part 5 of 6

represent a small rectangle hovering near the bottom of the rectangle we just drew. Once we've established this rectangle, we call `FrameOval`. `FrameOval` draws an oval whose leftmost, topmost, rightmost, and bottommost points overlap the specified rectangle. The rectangle itself isn't drawn, just the oval.

Then we change the pen color again, alter the rectangle record to be nested inside the oval, and call `PaintRect`. This fills the specified record with the current solid pen color.

Finally, we make a rectangle slightly inside that one and call `EraseRect`. `EraseRect` erases the specified rectangle to the port's background color (which is, by default, white).

Once our drawing is complete, we call `InitCursor` to show the cursor and do our same old Button loop to wait for a mouse click. Not a very friendly application—the window doesn't actually work—but you can see graphics on the super-hires screen!

Here are the additional resources needed for this program:

```
// Title of the window

resource rPString (1000) {" Untitled "};

// The rWindParam1 resource describes a
window.
```

```
resource rWindParam1 (1000) {
    $DDA5,                /* wFrameBits */
    1000,                  /* wTitle */
    0,                     /* wRefCon */
    {0,0,0,0},             /* ZoomRect */
    0x07FF0001,            /* Lined title */
    {0,0},                 /* Origin */
    {1,1},                 /* data size */
    {0,0},                 /* max height-width */
    {8,8},                 /* scroll ver hors */
    {0,0},                 /* page ver horiz */
    0,                     /* winfoRefcon */
    10,                    /* wInfoHeight */
    {30,10,183,282},       /* wposition */
    infront,               /* wPlane */
    nil,                   /* control list */
    $0A09                  /* input descriptions */
};
```

The `rPString` resource provides a Pascal format string (a length byte followed by the actual text) that we'll use as the window's title. The `rWindParam1` resource describes the window. We'll look at this in more detail next time.

This article is already longer than it's supposed to be, so next time, we'll make a program with working menus and see how to handle window events properly. We'll also talk more about `GrafPorts` and generally wrap things up, since next time will be the last article in my GSoft BASIC programming series.

---

*It's time for another year of...*

# Juiced.GS

The Apple II world's last remaining print publication is ready for another year of fun and excitement! Please join us for four more issues of news, reviews, interviews, and more. Subscriptions cost \$19 in the USA, \$24 in Canada, and \$27 elsewhere in the world. Send a check or money order with the below information to:

**Gamebits / Attn: Juiced.GS / P.O. Box 703 / Leominster, MA / 01453-0703**

Name: \_\_\_\_\_

Address: \_\_\_\_\_ City: \_\_\_\_\_

State/Province: \_\_\_\_\_ Zip: \_\_\_\_\_ Country: \_\_\_\_\_

Email: \_\_\_\_\_

**Or you can use your credit card to order online!**

**Visit <https://juiced.gs/subscribe/>**

# Bringing it all together

Welcome to the final chapter of our GSoft Basic programming saga. In this installment, we're going to continue our discussion of Toolbox programming. Keep in mind that our purpose isn't so much to learn Toolbox programming as it is to learn how GSoft BASIC works with the Toolbox.

### GETTING A HANDLE ON MEMORY

The Apple IIGS computer can have up to 8 megabytes of memory; that's quite a lot to keep track of. In order to remove the burden of managing memory from the application programmer, the Apple IIGS provides the Memory Manager tool set. This tool set handles the task of tracking memory use, and of allocating memory for use by applications, desk accessories, and other software.

When a program needs some memory, it asks the Memory Manager to allocate a block of memory. This is done by calling the `NewHandle` function, which returns a handle to a block of memory. A handle is a special pointer to a pointer to a block of memory. In addition, a handle includes special information about the memory block. For example, if the block of memory is locked down so it can't be moved, there is a special bit flag set to indicate this. There are a number of options available for handles; however, we don't have time to investigate them all.

The key point you should take away from this is simple: any memory you use when programming the Apple IIGS must be in a valid handle. If it's not, bad things will happen. In particular, when you load resources from disk they will be loaded into new handles in memory.

### A PORT IS A PORT OF COURSE OF COURSE

On the Apple IIGS, all graphics drawing performed using the QuickDraw tool set is done into GrafPorts. A GrafPort is a conceptual viewport onto a canvas. The content area of the window is a GrafPort, the menu bar is a GrafPort, the desktop is a GrafPort, and so forth. The vast majority of QuickDraw functions are used to create, alter, and draw in GrafPorts.

For example, if you have a 20,000 page document in a word processor, you can't possibly see the entire thing on your screen and the same time. Instead, you see a small portion of your document, and use a scroll bar to adjust what part of the document you're looking at. The

content area of your window, which is a GrafPort, is updated to reflect the portion of the document you're able to see. This is a lot like taking along a roll of paper and moving it back and forth behind your screen.

We did some drawing into a GrafPort last time, when we drew rectangles and ovals into a window. For most programs, this brief description of GrafPorts is really all you need to know. However, if you want to do more involved programming, you should look at the QuickDraw chapter in the Apple IIGS Toolbox Reference manual.

### THE MAIN EVENT

As we saw, briefly, last time, Apple IIGS desktop programs operate by using a central event loop, which processes user interaction events and responds appropriately. In some cases, an event has a very specific universally-defined meaning. When an event occurs, a special structure called an `EventRecord` is created to represent that event. This `EventRecord` is then inserted into a list of pending it one called the event queue. Applications that need to respond to events should periodically check for the presence of an event in the event queue. This is typically done once each time through the main event loop. Whenever an event is found in the event queue, the application responds to it.

There are, basically, two ways to obtain events from the event queue. The first is to call the `GetNextEvent` function, which fetches the next event from the queue, returning it to the application, and removing it from the queue.

The application is then responsible for figuring out which event occurred, how best to handle it, and actually handling it. This can be a fairly complicated task.

Fortunately, there's an easier way to do it. This easier way is brought to Apple IIGS programmers through the `TaskMaster` function. This function, which does not exist on the Macintosh, automatically fetches the next event from the event queue and performs much or all of the task of handling the function. The degree to which the operation is handled varies depending on the particular event.

If `TaskMaster` is only able to partially handle the event, it will return to your application information that you'll need to finish the job. The amount of work you can

---

# Programming: GSoft BASIC, part 6 of 6

save by using TaskMaster is so great that we won't even bother looking at how to do this using GetNextEvent.

Let's take a look at the EventRecord structure in GSoft BASIC:

```
type eventRecord
    eventWhat as integer
    eventMessage as long
    eventWhen as long
    eventWhere as point
    eventModifiers as integer
; TaskMaster fields
    taskData as long
    taskMask as long
    lastClickTick as long
    ClickCount as integer
    TaskData2 as long
    TaskData3 as long
    TaskData4 as long
    lastClickPt as point
end type
type eventRecPtr as pointer to
eventRecord
```

The eventWhat field indicates what type of event occurred. The eventMessage field contains event-specific data. For example, if the event it describes is a keyboard event, the key that was pressed or released is identified by this field. The eventWhen field indicates the time at which the event occurred, in 60ths of a second since the computer was started up. eventWhere indicates the position of the mouse cursor when the event occurred (even if it's not a mouse event), and eventModifiers indicates the state of the keyboard modifier keys, such as the Apple, option, and shift keys.

The meanings of the remaining fields vary depending on the particular event that occurred. We'll discuss them as we discuss those events, assuming we get to them. Since this lesson is only intended to be an introduction to GSoft BASIC Toolbox programming, we won't get into very much detail about all of these events.

The simplest event loop will simply consist of a while loop that exits when it's time to quit the application. Inside this loop is a call to TaskMaster, after which there might be code to complete the handling of certain events. That's honestly all there is to it!

## LET'S WRITE SOME CODE!

For our final lesson, we're going to write a simple framework Toolbox application which covers all the

basics. It'll have a menu bar, a simple about box using an alert window, a fully-functional window, and an event loop to tie it all together. It won't necessarily be a particularly useful application, but it'll be a good starting point for your own projects in the future.

This program and its Rez source are available on the Shareware 2000/Winter two-disk set, now part of the *Friends For Life* CD, available from *Juiced.GS*.

Before the actual code starts, we have a number of global variable declarations, some constants, and a couple of new types to define. Let's look at that first.

```
dim done as integer          : ! tells if
the program should stop
dim userID as Integer        : ! our user ID
dim event as integer         : ! event #;
returned by TaskMaster
dim lastEvent as eventRecord : ! last
event returned in event loop
dim startStopParam as long   : ! tool
startup/shutdown information
```

The done variable will be used as a boolean flag; normally false, we'll set it to true when the user chooses the Quit option in the File menu. userID will contain the Memory Manager user ID assigned to the program. event will be the task code returned by the TaskMaster call, and lastEvent will be the event record for that event. startStopParam will contain the result from the StartUpTools function, just like we learned last time.

```
type documentStruct
    : ! information about our
document
    after as pointer to
documentStruct : ! next document
    wPtr as GrafPortPtr : !
window pointer
    wName as String      : !
window name
    onDisk as Boolean     : !
does the file exist on disk?
    drawProc as procPtr  : !
pointer to drawing proc
    r as Rect            : !
just a random rectangle
    color as Integer      : !
color to draw in
end type
type documentPtr as pointer to
documentStruct : ! document pointer
```

---

# Programming: GSoft BASIC, part 6 of 6

These types will be used to track our open windows. Each time we open a window, we'll create a document record using this type. It'll be a linked list (which we've also talked about before, hooray!). Each window will have one rectangle in it, of a random color.

```
dim documents as documentPtr      : !
our documents
dim dPtr as documentPtr
```

Documents is the head pointer for our linked list of document windows. dPtr will be used by the main program for accessing documents.

```
const appleAbout = 257
const fileClose = 255
const fileQuit = 256
const fileNew = 260
const fileOpen = 261
```

These constants are the ID numbers of the menu items in our menu bar.

Next we load the tools we need, set up our user interface, and prepare to enter the main event loop.

```
userID = MMStartUp
startStopParam = StartUpTools(userID,
refIsResource, 1)
if toolerror <> 0 then
    print "Unable to load tools."
end
end if

call InitGlobals
call InitMenus
lastEvent.taskMask = $1FFF
ShowCursor
InitCursor

done = 0
```

The program begins by initializing the Memory Manager and getting our user ID, then starting up the Toolbox like we did last time. Then we call the InitGlobals subroutine to initialize global variables, and InitMenus to load and set up our menu bar. We'll look at these subroutines in a bit.

We then initialize the taskMask field of the lastEvent EventRecord to \$1FFF. The taskMask is how we tell TaskMaster what events to automatically handle. Each bit in this 32-bit value is a boolean flag that lets us turn on or off a particular TaskMaster

feature. You can see a complete list of these bits and what they do in the Window Manager chapter of the Toolbox Reference, volume 3.

Then we call InitCursor to initialize the cursor to the arrow, and we set done to 0 (indicating that it's not time to quit yet). Now it's time to enter the main event loop.

```
while not done
    event = TaskMaster(everyEvent,
lastEvent)
    select case event
        case wInSpecial, wInMenuBar
            call
HandleMenu(lastEvent.taskData, done)
        case wInGoAway
            dPtr =
FindDocument(GrafPortPtr(lastEvent.taskData
))
            call CloseDocument(dPtr)
    end select
wend
```

The event loop is a while loop that repeats as long as done is zero. First it calls TaskMaster. everyEvent indicates that when TaskMaster calls GetNextEvent to fetch the next user event, it should look for all events. If you only wanted it to handle key presses, you could say keyDownMask+autoKeyMask, but you almost always want to handle them all.

TaskMaster returns a special task code indicating what, if anything, your application needs to do. The lastEvent EventRecord will be filled out by TaskMaster with information about the event, plus any additional information you need to finish the job.

We then use a select-case statement to look for task codes we need to respond to.

The wInSpecial task code indicates that the user picked a special menu item. The special items are the ones that have predefined item numbers. These are:

```
250 = Undo
251 = Cut
252 = Copy
253 = Paste
254 = Clear
255 = Close
```

Desk accessories all have item numbers between 1 and 249, so your application can't use those numbers.

---

# Programming: GSoft BASIC, part 6 of 6

More on menus when we talk about the InitMenus subroutine.

The wInMenuBar task code indicates that one of your application's menu items was picked.

If either wInSpecial or wInMenuBar was returned by TaskMaster, we call HandleMenu, a function we define to process the menu selection. We pass lastEvent.taskData, which contains the menu and menu item numbers selected. We also pass in the done variable, which may be changed if the user picked the Quit option from the File menu.

The wInGoAway task code is returned if the user clicked the close box in a window. Actually, there's a tad more to it than that. If you hadn't noticed, clicking the close box doesn't actually close a window. Clicking and releasing the mouse button in the close box does. If you click in the close box, then move the mouse out of the box before letting go of the button, nothing happens. TaskMaster deals with this for you, and only returns wInGoAway if this happens. Do you love TaskMaster yet?

If TaskMaster returns wInGoAway, the taskData field is a pointer to the GrafPort for the window the user wants to close. We pass this to our FindDocument function, which returns a pointer to the document record for the window. We can then call our CloseDocument function to close the window and dispose of the associated data.

There are of course a number of other task codes, including wInControl, which is returned if the user clicked in a control (such as a button or checkbox).

Once the main event loop is exited (by the user choosing to Quit), we shut down the tools and end the program:

```
ShutDownTools(1, startStopParam)
MMShutDown(userID)
end
```

Let's see how we create the menu bar used in this program. The menu bar is created by the InitMenus subroutine:

```
sub InitMenus
    dim height as integer          : !
height of the largest menu
    dim m as menuBarHandle         : ! our
menu bar's handle
```

```
! create the menu bar
m = NewMenuBar2(refIsResource, 1,
NIL)
SetSysBar(ctlRecHndl(m))
SetMenuBar(NIL)
! add desk accessories
FixAppleMenu(1)
! draw the completed menu bar
height = FixMenuBar
DrawMenuBar
end sub
```

We have two local variables: height will be the height of the menu bar, and m will contain a handle to the menu bar.

To create the menu bar, we call the Menu Manager function NewMenuBar2. This function uses a menu bar template to construct a new menu bar. In this case, we want to load the menu bar from the menu bar template resource with the ID 1. The last argument to NewMenuBar2 is a window pointer; if you pass a GrafPort pointer, you can put the menu bar inside a window. There's actually a good deal more to it than that, but that's another story. Passing NIL means you want to create a system menu bar. A system menu bar is one that sits at the top of the screen, like almost every menu bar you've seen on the IIGS.

NewMenuBar2 creates a menu bar, but doesn't make it visible, or even available for use. You then have to call SetSysBar to make the new menu bar the current system menu bar. You have to cast the handle to a ctlRecHndl type for historical reasons.

Next, call SetMenuBar. This selects which menu bar the Menu Manager functions will work with. Since it's possible to have multiple menu bars (a system bar and window menu bars), you have to let the Menu Manager know which one you want to talk to. The FixAppleMenu function installs the NDAs into the Apple menu. The argument indicates the menu ID of the Apple menu. The NDAs are added after any other menu items you've already put in the menu.

Then we call the FixMenuBar function, which computes the height of the menu bar, which can vary depending on what the menu bar font is. In reality, due to limitations in the system software, this will be 13 pixels almost all the time, but you should never assume this (that would be very poor programming style). FixMenuBar returns the height of the menu bar.

Finally, we call DrawMenuBar to draw the new menu bar.

# Programming: GSoft BASIC, part 6 of 6

Here are the resources used for the menu bar and its menus and menu items:

```
resource rMenuBar (1) { /* the menu bar
*/
{
    appleMenu, /* resource numbers for
the menus */
    fileMenu,
    editMenu
};
};

resource rMenu (appleMenu) { /* the
Apple menu */
    appleMenu, /* menu ID */
    refIsResource*menuItemRefShift /*
flags */
    + refIsResource*itemRefShift
    + fAllowCache,
    appleMenu, /* menu title resource
ID */
    {appleAbout}; /* menu item resource
IDs */
};

resource rMenu (fileMenu) { /* the File
menu */
    fileMenu, /* menu ID */
    refIsResource*menuItemRefShift /*
flags */
    + refIsResource*itemRefShift
    + fAllowCache,
    fileMenu, /* menu title resource
ID */
    {
        /* menu item resource
IDs */
        fileNew,
        fileOpen,
        fileClose,
        fileQuit
    };
};

resource rMenu (editMenu)
{
    /* the Edit menu */
    editMenu, /* menu ID */

    refIsResource*menuItemRefShift /*
flags */
    + refIsResource*itemRefShift
    + fAllowCache,
```

```
    editMenu, /* menu title resource
ID */
{
    /* menu item resource
IDs */
    editUndo,
    editCut,
    editCopy,
    editPaste,
    editClear
};

resource rMenuItem (editUndo){ /* Undo
menu item */
    editUndo, /* menu item ID */
    "Z","z", /* key equivalents */
    0, /* check character */
    refIsResource*menuItemRefShift /*
flags */
    + fDivider,
    editUndo /* menu item title resource
ID */
};

resource rMenuItem (editCut){ /* Cut
menu item */
    editCut, /* menu item ID */
    "X","x", /* key equivalents */
    0, /* check character */
    refIsResource*menuItemRefShift, /*
flags */
    editCut /* menu item title resource
ID */
};

resource rMenuItem (editCopy){ /* Copy
menu item */
    editCopy, /* menu item ID */
    "C","c", /* key equivalents */
    0, /* check character */
    refIsResource*menuItemRefShift, /*
flags */
    editCopy /* menu item title resource
ID */
};

resource rMenuItem (editPaste) { /*
Paste menu item */
    editPaste, /* menu item ID */
    "V","v", /* key equivalents */
    0, /* check character */
    refIsResource*menuItemRefShift, /*
flags */
```



# Programming: GSoft BASIC, part 6 of 6

```
editPaste /* menu item title resource
ID */
};

resource rMenuItem (editClear) { /*
Clear menu item */
editClear, /* menu item ID */
"", "", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift, /*
flags */
editClear /* menu item title resource
ID */
};

resource rMenuItem (fileNew) { /* New
menu item */
fileNew, /* menu item ID */
"N", "n", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift, /*
flags */
fileNew /* menu item title resource
ID */
};

resource rMenuItem (fileOpen)
{ /* Open menu item */
fileOpen, /* menu item ID */
"O", "o", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift /*
flags */
+ fDivider,
fileOpen /* menu item title resource
ID */
};

resource rMenuItem (fileClose) { /*
Close menu item */
fileClose, /* menu item ID */
"W", "w", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift /*
flags */
+ fDivider,
fileClose /* menu item title resource
ID */
};

resource rMenuItem (fileQuit) { /* Quit
menu item */
fileQuit, /* menu item ID */
"Q", "q", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift, /*
flags */
fileQuit /* menu item title resource
ID */
};

resource rMenuItem (appleAbout) { /*
About menu item */
appleAbout, /* menu item ID */
"", "", /* key equivalents */
0, /* check character */
refIsResource*itemTitleRefShift /*
flags */
+ fDivider,
appleAbout /* menu item title resource
ID */
};

resource rPString (appleMenu) {"@"};
resource rPString (fileMenu) {" File
"};
resource rPString (editMenu) {" Edit
"};
resource rPString (editUndo) {"Undo"};
resource rPString (editCut) {"Cut"};
resource rPString (editCopy) {"Copy"};
resource rPString (editPaste)
{"Paste"};
resource rPString (editClear)
{"Clear"};
resource rPString (fileNew) {"New"};
resource rPString (fileOpen) {"Open"};
resource rPString (fileClose)
{"Close"};
resource rPString (fileQuit) {"Quit"};
resource rPString (appleAbout) {"About
RealApp..."};

HANDLING MENU SELECTIONS

The HandleMenu subroutine reacts to the user
selecting an item in the menu bar. The menu item
number the user picked is in the low word of the
taskData field of the EventRecord, which is passed into
this function. Based on that value, we respond.

sub HandleMenu (taskData as long, done
as integer)
shared appleAbout, fileQuit,
fileOpen, fileNew
shared fileClose
```

---

# Programming: GSoft BASIC, part 6 of 6

```
dim w as GrafPortPtr
dim dPtr as documentPtr

select case LoWord(taskData)
case fileQuit
    done = 1
case appleAbout
    call DoAbout
case fileClose:
    w = FrontWindow
    dPtr = FindDocument(w)
    call CloseDocument(dPtr)
case fileNew:
    dPtr = NewDocument("
Untitled ")
case fileOpen:
    dPtr = NewDocument(" Opened
Document ")
end select
HiliteMenu(0, HiWord(taskData))
end sub
```

If the user picked the Quit option, we set done to 1, indicating that the program should exit. If they picked the About RealApp option, we call DoAbout, a subroutine that will display an about box.

If the user picked the Close option, we call the Window Manager function FrontWindow to get a GrafPort pointer to the frontmost window, then call FindDocument to get a pointer to the document record for that window, then call CloseDocument to close that window.

In this simple program, the New and Open options do the same thing: they call NewDocument, a function that creates a new window. They pass in a title for the new window.

You can handle additional menu items by simply adding them to this select-case statement.

Tell the world about yourself

```
sub DoAbout
    const alert ID = 1
    dim b as integer

    b = AlertWindow(awCString
+awResource, NIL, alertID)
end sub
```

This simply calls AlertWindow. The first parameter is a flag word. In this case, the only flags we're

specifying are the ones defining how substitution strings are referenced (as CString resources). The second parameter is a pointer to an array of substitution strings; we don't have any, so we pass NIL. The last one is the ID of other AlertString resource, 1.

AlertWindow will replace the characters "\*"n" with the nth string in the substitution array you specify ("\*0" is replaced with the first string, "\*1" is replaced with the second, and so forth). This can be very handy, and the FlagError subroutine will use it.

AlertWindow returns the number of the button that was clicked. Button 0 is the leftmost button, and the number is one larger for each button to the right. There can be up to three buttons in the alert; our about box has only one.

The rAlertString resource for the about box looks like this:

```
resource rAlertString (1) {
    "43/"
    "RealApp.bas\n"
    "A sample BASIC program using
the Toolbox.\n\n"
    "By Eric Shepherd\n"
    "/^#0\,$00;
};
```

The format of an rAlertString is beyond the scope of this article.

## CREATING A DOCUMENT

```
function NewDocument(wName as String) as
documentPtr
    shared documents

    const wrNum = 1001
: ! window resource number
    dim dPtr as documentPtr
: ! new document pointer
    dim s as string

    allocate(dPtr) : ! allocate the
record
    if dPtr <> NIL then
        dPtr^.onDisk = false
: ! not on disk
        dPtr^.wName = wName
: ! the name
```

---

# Programming: GSoft BASIC, part 6 of 6

```
dPtr^.r.left = rnd(1) * 300
dPtr^.r.right = rnd(1) *
300 + 321
dPtr^.r.top = rnd(1) * 90
dPtr^.r.bottom = rnd(1) *
90 + 101
dPtr^.color = rnd(1) * 15
dPtr^.drawProc =
AllocateProc(DrawContents)
s = chr$(len(wName)) +
wName
dPtr^.wPtr =
NewWindow2(pStringPtr(@s), 0,
dPtr^.drawProc, NIL, refIsResource, wrNum,
rWindParam1)
if dPtr^.wPtr = NIL then
call FlagError("Toolbox
error in NewWindow2 call", toolerror)
: ! handle a window error
dispose(dPtr)
dPtr = NIL
else
dPtr^.after = documents
: ! put the document in the list
documents = dPtr
end if
else
call FlagError("Out of
memory", 0) : ! handle an out of memory
error
end if

NewDocument = dPtr
end function
```

NewDocument begins by allocating memory for the new document record, using the allocate statement. If this is successful (dPtr isn't NIL), the record can be safely filled out with the appropriate data, and the window can be opened.

The fields of the document record are filled out; onDisk is a flag indicating whether or not the document needs to be saved to disk; we aren't going to use it in this program, but we do initialize it to false, indicating that it hasn't been saved yet. The wName field is the document's name. I suggest using this field for a pathname, but in this case we're storing just the title of the window, since we won't be doing file saving in this program.

Then the rectangle and color are randomly created, using the BASIC rnd() function to generate random

numbers; rnd() generates a random number between 0 and 1—we multiply and add to scale the value into the range we want.

When the Window Manager determines that a window needs to be updated on screen—for example, if it's being brought in front of another window that was obscuring part or all of it—it calls a content definition procedure function (often called a content defproc or drawproc) to draw the window's contents.

This function is called directly by the Toolbox, and your application should provide one. Because your program is in BASIC, you can't directly specify a drawproc to be used.

So GSoft BASIC provides the AllocateProc function, which constructs an assembly language wraparound for a GSoft BASIC function. This lets you use GSoft BASIC code in callback functions of this type. So we create the drawproc for our window by calling AllocateProc; the drawproc will be our DrawContents subroutine. This is stashed in the document record as well, since we'll have to dispose of it when the window is closed.

Then we need to create the string needed for naming the window.

We have to pass a Pascal string to the NewWindow2 call, and GSoft BASIC strings aren't Pascal strings. So we need to create one. We do this by creating a string, s, in which the first character is the character corresponding to the ASCII code that's the length of the window's title we want to use, and the rest of the string is the title itself. Then we can pass this to NewWindow2, cast to a pStringPtr type.

If NewWindow2 returns NIL, it's an error and we call FlagError to report it to the user; we then dispose of the document record. Otherwise, we add the new record to the beginning of the window list.

If we weren't able to allocate the document record, we call FlagError.

NewDocument then returns the new document record.

The resource for the window looks like this:

```
resource rWindParam (1001) {
$DDa5,          /* wFrameBits */
nil,            /* wTitle */
0,              /* wRefCon */
}
```

---

# Programming: GSoft BASIC, part 6 of 6

```
{0,0,0,0},          /* ZoomRect */
nil,                /* wColor ID */
{0,0},             /* Origin */
{200,640},         /* data size */
{200,640},         /* max height-width */
{8,8},            /* scroll ver hors */
{30,30},          /* page ver horiz */
0,                /* winfoRefcon */
10,              /* wInfoHeight */
{30,10,183,602},  /* wposition */
infront,         /* wPlane */
nil,            /* wStorage */
$0000          /* wInVerb */
};
```

## DRAWING THE CONTENTS

```
sub DrawContents
    dim dPtr as DocumentPtr
    PenNormal : ! Set default
drawing pen

    dPtr = FindDocument(GetPort) : !
Find the document
    to draw
        SetSolidPenPat(dPtr'.color) : !
Set pen color
    PaintRect(dPtr*.r) : ! Draw the rectangle
end sub
```

The drawproc in this example is very simple. It simply uses FindDocument to determine the document record for the window being drawn (it calls the QuickDraw function GetPort to determine the current GrafPort), then sets the pen color and paints the window's rectangle.

Of course in your own program you'd replace this code with whatever's appropriate. If you have controls in your window, you might call DrawControls in the Control Manager to refresh them, for example.

## CLOSING A DOCUMENT

```
sub CloseDocument(dPtr as documentPtr)
    shared documents

    dim lPtr as documentPtr : !
pointer to the previous doc.
    if dPtr <> NIL then
```

```
CloseWindow(dPtr^.wPtr) : !
close the window
    if documents = dPtr then
        documents =
dPtr'.after : ! ...dPtr is the first
document
    else
        lPtr = documents : ! ...dPtr
isn't the first document
        while lPtr^.after <> dPtr
            lPtr = lPtr^.after
        wend
        lPtr^.after = dPtr^.after
    end if
    DisposeProc(dPtr^.drawProc) :
! dispose of the drawproc
    dispose(dPtr) : ! dispose of
the document record
end if
end sub
```

CloseDocument accepts as input a document record pointer. It first checks to be sure it's not NIL. Then it closes the window by calling the CloseWindow Window Manager function. It then removes the record from the window list, using code much like we've seen when we were talking about linked lists before. The only difference is that we use the DisposeProc statement to dispose of the drawproc before we dispose of the document record itself.

## MATCHING A GRAFPORT TO DOCUMENT RECORD

Sometimes you'll know a GrafPort pointer, but you need to find the corresponding document record.

```
function FindDocument(wPtr as
GrafPortPtr) as documentPtr
    shared documents

    dim done as Boolean a : !
used to test for loop termination
    dim dPtr as documentPtr : !
used to trace the document list

    dPtr = documents
    if dPtr = NIL then
        done = true
    else
        done = false
    end if
```

---

# Programming: GSoft BASIC, part 6 of 6

```
while not done
    if dPtr^.rwPtr = wPtr
then
        done = true
    else
        dPtr = dPtr^.after
        if dPtr = NIL then
            done = true
        else
            done = false
        end if
    end if
wend

FindDocument = dPtr end function
end function
```

This code simply walks the document list, comparing the GrafPort pointer wPtr to the GrafPort pointer in each document record.

When a match is found, that document pointer is returned. If no match is found, the function returns NIL (the after field in the last record in the list is NIL, so when the list runs out, this is returned).

## REPORTING ERRORS

```
sub FlagError(msg as String, code as Integer)

    const errorAlert = 2000
    dim substArray(1) as StringPtr

    dim errStr as String
    dim.b as Integer

    substArray(0) = msg + " (Error
$" + hex$(code) + ")"
    b = AlertWindow(awCString
+awPointer, ptr(@substArray(0)),
errorAlert)
end sub
```

This simple error handler subroutine uses an rAlertString that simply contains "\*"0" and builds a string to be used in the substitution array. This string consists of a message passed in by the caller and the text "(Error \$errnum)" where errnum is the hexadecimal version of the error code passed into the subroutine.

The alert has only one button, so the result of AlertWindow can be ignored.

The resource looks like this:

```
resource rAlertString (errorAlert) {
    "32/"
    "*"0"
    "/A^#0\ $00";
};
```

## PUTTING IT ALL TOGETHER

You now have everything you need to build this program. You can reuse the tool startup table resource from last time. Just change the mode320 flag to mode640 so your program will run in 640 mode instead of 320 mode.

## THAT'S ALL, FOLKS!

That's the end of the GSott BASIC series. I'll be continuing to write programming articles for Max, although at this point I don't know what they'll be yet. I have a challenge to issue to you.

The Apple II world needs new software. With the knowledge gained by reading these articles, you have the information needed to create some. Software doesn't have to be flashy to be useful.

You could write a math drill program for children. Just generate random numbers based on the difficulty level the child needs, and ask them questions like "What's 5+10?" and the like.

Once you have time basic drill, you could even augment it with features to make it more fun for the child. But even the basic drill program could be handy for a parent trying to find something to do with the IIGs they picked up at the garage sale for their kids.

Or write a simple program for cataloging videotapes, using a random-access file. Or a program to automatically scan disks, generating an index of the tiles on them (we briefly talked about how to read directories once).

There are a thousand possibilities, and my challenge to you is to write them and release them to the public.

If you're not sure they're bug-free, find someone you trust to test it for you. If you get stuck, or have questions, I'd be happy to try to help out—just e-mail sheppy@sheppyware.net. I can't promise I can answer every question you might have, but I'll try.

The Apple II world needs new software. You have the tools and the knowledge to help. Please do.