



STUDIO BASIC

Tap the power of the newest II—with a good idea and an idle weekend, you can develop your own GS solutions for home, school, and career.

By **JOE ABERNATHY**

UNTIL RECENTLY THE MOST POWERFUL Apple II has also been the most difficult II to program. But don't let the IIGS intimidate you; let it challenge you. Thanks to several third-party language developers, the full creative scope of the IIGS is now available to you as a BASIC programmer.

DEVELOPMENT PLATFORMS

Before you even write your first line of code, you'll need to determine which compiler to use. Unlike built-in Applesoft, the 8-bit *interpreter* that translates BASIC into machine language line by line, *compilers* translate an entire BASIC program into machine language. Currently, you can choose among four commercial BASICs. (A fifth is scheduled for release this summer.) As you read the following rundown, keep your ultimate programming goals in mind. That should be the determining factor in selecting the compiler that's best for you.

AC/BASIC. This is the fastest way to program your IIGS. Intuitive one-word commands replace traditional Toolbox access, making desktop programming readily accessible even to beginners. AC/BASIC is good for any casual custom programming, but lacks the speed or flexibility to produce high-level applications.

Any Microsoft-compatible BASIC compiler can share AC/BASIC source code, so you can adapt a well-designed AC program written on

your IIGS for another brand of computer in just a few hours.

GS BASIC. No longer produced or supported, this compiler is still available through the Apple Programmers and Developers Association (APDA) at Apple Computer.

Micol Advanced BASIC. The new kid on the block is being touted as a latter-day Applesoft, with Toolbox support and structure added. This compiler, which doesn't require in-depth instruction, lets you merge special GS capabilities with programs written on older IIs.

Micol is the only BASIC that can produce classic desk accessories (CDAs, like the GS control panel), a feature that should be available in every language. It also supports structured programming considerably more advanced than that of other compilers.

Micol BASIC isn't targeted at those who want to produce "genuine" IIGS desktop applications. This capability exists in theory, but not in practice. Although Micol Systems plans significant enhancements, for now you'll have to weigh its comparatively limited abilities against its price, the highest of any IIGS BASIC.

ORCA/BASIC. ByteWorks plans to release ORCA/BASIC this summer. You'll be able to use this product with other ORCA languages in a single program. Reportedly, ORCA/BASIC will incorporate a number of other powerful features not yet available in IIGS BASIC.

TML BASIC. This language produces the fastest programs and has the best manual. It em-

plays a traditional Toolbox interface: You'll have complete tool access, but you must use the Toolbox reference manuals in conjunction with the compiler.

TML BASIC is the best of the lot for advanced application programming. Its power, together with its thorough implementation of structured programming and the GS Toolbox, makes it shine.

Which language should you buy? Your goals for the coming weekend are only half the story. Consider also what you hope to be doing six months from now, and choose a product that will grow with you.

If you prefer traditional Applesoft and a text-based interface, or want to write CDAs, Micol Advanced BASIC will add the most to your efforts. Choose TML if you want to design a large desktop application. AC is the best for spare-time programmers who want to present their work under the desktop metaphor.

STRUCTURED PROGRAM DESIGN

A structured program is one you write in terms of "building block" subroutines. Structured programming requires more learning and planning than does simple hacking at the keyboard, but you must embrace it to realize the GS' abilities. Soon, you'll wonder how you got by without it.

The theory behind modular programming is that you can define even a large program as a series of low-level tasks. When you first write a procedure, its simple scope makes it easy to program and debug. You can then use the procedure, if designed properly, in every program that needs its services. You'll eventually develop a personal library of procedures that might handle anything from graphics to database manipulation.

Looking at the big picture, structured design adds elegance to program flow. Procedures replace the GOTO statement, for instance. (Think of them as a latter-day version ►

of GOSUB.) Eliminating GOTO forces you to use solid program logic, and has the side benefit of making programs self-documenting.

Modular design is based on control structures—language features that let you plan block design and program flow. Control structures are a series of statements in the form of IF/THEN/ELSE, DO/UNTIL, DO/WHILE, and so on. This differs from Applesoft BASIC, in which a program can move only forward in a straight line (line 10, line 20, and so on).

The best way to grasp these structures is to see them in action:

```
IF Online% = Nil% THEN
  PROC DoDialOut (Phone%)
ELSE
  PROC DoLogon (Macro%)
END IF
WHILE NOT EOF (FileIndex%)
  PROC ReadLine
  PROC ShowLine
WEND
DO
  PROC ReadClient
UNTIL ClientNum% = NumClients%
DO
  PROC ReadDataStream
  CharCount% = FN CountEm
WHILE NOT EOL (1)
CASE TestScore% OF
  100 : PROC SetCurve
  90 : PROC DoA
  80 : PROC DoB
  70 : PROC DoC
  60 : PROC DoD
ELSE
  PROC DoFlunk (Student$)
ENDCASE
```

Although each compiler offers a different set of control structures, there will always be a way to make the program do what you want it to. For instance, only Micol BASIC offers the CASE statement, but you can mimic it with the IF/THEN/ELSE of AC and TML.

In the examples above, you can see that procedures, identified by PROC, and functions (procedures that calculate and return a value) are an integral part of structured program flow. Here are examples of what some of the definitions of the fictitious procedures used above might look like:

```
PROC DoDialOut (Phone%)
```

Listing 1. inCider.Shell.

```
* File: inCider.Shell
* By Joe Abernathy
* (C)1989, Joe Abernathy. All Rights Reserved.
* Compiler: AC/BASIC for the Apple IIGS.
* Compile this shell with the "no standard menus" option selected.

Portions of this program include material copyrighted (C) by Absoft
Corp. 1988. Used with permission. All other copyrights acknowledged.

* Customize the ABOUT box:
* $About "inCider Shell, By Joe Abernathy"

GOSUB DoSetup
DoMenu
ON MENU GOSUB menuproc
MENU ON

main:
  GOTO main

menuproc:
  menunum = MENU(0)
  itemnum = MENU(1)
  IF menunum = 1 THEN
    IF itemnum = 1 THEN
      GOSUB 10
    ELSEIF itemnum = 2 THEN
      GOSUB 20
    ELSEIF itemnum = 3 THEN
      GOSUB 30
    ELSEIF itemnum = 4 THEN
      GOSUB 40
    ELSEIF itemnum = 5 THEN
      GOSUB 50
    ELSEIF itemnum = 6 THEN
      GOSUB 60
    END IF
  END IF
  RETURN

* This routine generates the standard new file dialog. It does not
* do anything with the selected file. In a program, you would use the
* which$ value as a parameter to the editing routines in your library.

10:
  f$ = FILES$(0,"DEFAULT.NAM")
  IF f$="" THEN RETURN
  which$ = f$
  MENU "Should be called after working with the file."

RETURN

* This routine generates the standard get file dialog. It does not
* do anything with the selected file. In a program, you would use the
* which$ value as a parameter to the editing routines in your library.

20:
  f$ = FILES$(1,"DIRTXT")
  IF f$="" THEN RETURN
  which$ = f$
  MENU "This should actually be called after working with the file."

RETURN

30:
  KillFiles
  MENU
  RETURN

* DELETE a file
* sub from inCider.Tools
* Unhighlight menu bar
```

```
PROC InitModem
PROC OffHook
PROC DialIt (Phone%)
END PROC
PROC ReadLine
```

```
LOCAL x$
tmpline$ = ""
WHILE NOT EOL (FileIndex%)
  x$ = GetChar
  tmpline$ = tmpline$ + x$
```

```

'-----
40:                                ' PRINT a file
f$ = "null"                        ' Force first While loop ..
WHILE f$ <> ""
    f$ = FILES$(1,"TXTSRC")
    IF f$ <> "" THEN
        PrintFile(f$)
    END IF
WEND
MENU                                ' Unhighlight menu bar
RETURN

'-----

50:                                ' Type a file to screen
f$ = "null"                        ' Force first While loop ..
WHILE f$ <> ""
    f$ = FILES$(1,"TXTSRC")
    IF f$ <> "" THEN
        TypeFile(f$)
    END IF
WEND
MENU                                ' Unhighlight menu bar
RETURN

'-----

60:                                ' QUIT
MENU                                ' Un-inverse menu bar
END

'-----

DoSetup:                          ' Set up program globals
top = 26                          ' Screen dimensions:
left = 4
bottom = 197
right = 618
WindEx = 2
flag = 0
filenum = 1
RETURN

'-----

SUB DoMenu                        ' Create menu bar
MENU 1,0,1,"File"                ' The FILE menu
MENU 1,1,1,"New"                  ' is it for now.
MENU 1,2,1,"Edit"
MENU 1,3,1,"Delete"
MENU 1,4,1,"Print"
MENU 1,5,1,"Type"
MENU 1,6,1,"Quit"
END SUB

'-----
' With structured programming, you will develop libraries of routines to
' handle various standard tasks. These must be attached to the program at
' compile time. This is done in AC/BASIC by putting appropriate INCLUDE
' directives at the end of the source code. You should create a data volume
' to hold these libraries. Below, specify for INCIDER.TOOLS the disk volume
' in which you have it installed.

'$INCLUDE "*/ASRC/BASIC/AC/DUMP/INCIDER.TOOLS"

'-----
' The End. (inCider.Shell)

```

WEND
END PROC

In the first instance, the procedure "dial out" is made up of smaller procedures that

instruct the program to "initialize the modem," "take the phone off the hook," and "dial."

To introduce structured programming on the IIGS, the first program you'll write is a shell for AC/BASIC (**Listing 1**) that shows how

to manage various aspects of the desktop, such as pull-down menus. You can reuse this shell in subsequent programs. In addition, I'll present a library of low-level software tools (**Listing 2**) you'll want to use in most of the programs you write later.

In coming issues, I'll use the shell as a platform on which you'll implement dialogs, user input/output, data management, graphics, sound, and more.

What if you're using a different compiler? If it's TML BASIC, send a self-addressed, stamped envelope to *inCider* and you'll receive free of charge the shell implemented in TML BASIC. (It's too lengthy to print here.) You can download the shells from *inCider's* BBS (603-924-9801), and you can also refer to the product information listed at the end of the column. Because Micol Advanced BASIC isn't oriented toward desktop programming, no shell is available for it.

I'll discuss each tool in the library individually in a moment, but you also can learn from examining the program as a whole:

- Each procedure handles a narrowly defined task you'll probably be able to use in a later program. Logical branching replaces the GOTO statement for directing program flow. This makes debugging easier, and lends a self-documenting quality to the program.

- Mnemonic variable names, as opposed to abstract numbers and letters, make your intent much clearer. Consistent indentation of inner loops and control structures makes the program easy to read.

- Procedures included in the *inCider.Tools* library are the low-level routines most applications need. On a more advanced level, you'll want to build libraries in a topical fashion—for instance, sound, graphics, or input/output.

- In the shell, notice that subroutines 40 and 50, which correspond to the menu choices Print and Type, must provide the parameters (the filename) the program will print or display. PrintFile and TypeFile are essentially dumb, but this quality makes them suitable for a wide variety of applications. Study their implementation in *inCider.Tools* with this in mind. Compare to KillFiles.

- In the Subroutine DoSetup, I maintain a list of global variables. Because BASIC has no formal variable-declaration feature as Pascal does, it's a good idea to adopt this convention to keep your global variables straight, even ►

though you don't need to.

- To add a capability to the shell, add a line to the DoMenu subroutine, a corresponding line in the MENUPROC routine, and a label to do the actual work. The ON MENU directive makes this program design possible; it activates event polling in which the system will automatically handle any selections made from the pull-down menus. Your manual explains this at length, but all you really need to know is how to make it work. No other compiler uses this methodology.

- The last line of inCider.Shell uses the \$INCLUDE directive to tie the inCider.Tools library to the main program. Note that this directive is written just as shown, including the REMark character.

- The inCider.Tools library is a set of utilities that'll be some of the first building blocks you'll need. When designing your own procedures, remember to keep things simple, and document each procedure thoroughly so that you don't have to wonder how to call it or modify it later.

Listing 2. inCider.Tools.

```

'-----
' File : inCider.Tools
' (C)1989, Joe Abernathy. All Rights Reserved.
' Compiler : AC/BASIC for the Apple IIGS.
'-----
' This library requires global variables declared in inCider.Shell.
'-----
' Procedure PrintFile -- Print a file.
' You must pass name of file to print in thefile$.
SUB PrintFile(thefile$)
  SHARED FileNum ' Global var
  FileNum = FileNum + 1
  OPEN thefile$ FOR INPUT AS FileNum ' Open file passed in thefile$
  OPEN "LPT1:PROMPT" FOR OUTPUT AS FileNum + 1 ' Open printer
  WHILE NOT EOF(FileNum)
    LINE INPUT #FileNum, a$
    PRINT #FileNum + 1, a$ ' Print line from thefile$
  WEND
  CLOSE #FileNum + 1
  CLOSE #FileNum
  FileNum = FileNum - 1
END SUB

```

```

'-----
' Procedure TypeFile -- Show a file on the screen.
' You must pass name of file to type in myfile$.
SUB TypeFile(myfile$)
  SHARED FileNum ' Global var
  FileNum = FileNum + 1
  DoWind(1)
  OPEN myfile$ FOR INPUT AS FileNum ' Open file passed in thefile$
  WHILE NOT EOF(FileNum)
    LINE INPUT #FileNum, a$
    PRINT a$ ' Print line
  WEND
  CLOSE #FileNum
  FileNum = FileNum - 1
  DitchWind
END SUB
'-----
' Procedure KillFiles -- Kill one or more files, using std dialog.
SUB KillFiles
  f$ = "null" ' Force first loop ...
  WHILE f$ <> ""
    f$ = FILES$(1) ' Value will be set to "" when
    IF f$ <> "" THEN ' Cancel is clicked, forcing
      KILL f$ ' the Delete loop to repeat ...
    END IF
  WEND
END SUB

```

Continued

INDIVIDUAL TOOLS

PrintFile (thefile\$). This procedure prints a file to the installed printer. It requires that you pass the name of a file as its parameter. In my example, I get the filename from the standard select-file dialog, then repeatedly call PrintFile until you select the Cancel button. No error checking is performed to ensure that a printer is on line.

This procedure uses the "device name" LPT1 to send output to a printer in slot 1. You can change the device name to SCRn, KYBD, COM1, or CLIP to send a file to the screen, keyboard, any serial device, or the clipboard. COM1 and LPT1 use the modem and printer ports on the back of the GS.

TypeFile (myfile\$). This procedure prints a file to the screen display. It works like PrintFile, requiring that you pass it the filename to type.

KillFiles. KillFiles generates its own standard file dialog, and kills any number of files until you select Cancel. I wrote it this way to provide ►

Continued

```

'-----
' Procedure MsgDialog(msg$,style%) -- Generate dialog with a text string,
'                                   and one or two buttons depending on
'                                   the value of style%. When this proc
'                                   detects OK, it ends. If Cancel was
'                                   clicked, the val cancel% is set to 1.
'-----

SUB MsgDialog(msg$,style%) SHARED
  cancel% = 0
  DoWind
  LOCATE 2,2
  PRINT msg$
  BUTTON 50,1,"OK",(left+20,bottom-160)-(left+120,bottom-145),1
  IF style% = 2 THEN
    BUTTON 51,1,"Cancel",(left+135,bottom-160)-(left+235,bottom-145),1
  END IF
  WHILE DIALOG (0)=0 ' Clear dialog queue ...
    WEND
  WHILE DIALOG(0)<>1 ' Wait for real event ...
    WEND
  x = DIALOG(1) ' Read the event
  IF x = 51 THEN cancel% = 1
END SUB

'-----
' Procedure DoWind(kind%) -- Open a window.
'                           kind% specifies the style number.
'-----

SUB DoWind(kind%) SHARED
  WindEx = WindEx + 1
  WINDOW WindEx,"",(left,top)-(right,bottom),kind%
END SUB

'-----
' Procedure DoDWind -- Generate a window for use with dialogs.
'-----

SUB DoDWind SHARED
  WindEx = WindEx + 1
  WINDOW WindEx,"",(left-5,top+10)-(right+5,bottom-100),2
END SUB

'-----
' Procedure DitchWind -- Close a previously opened window, decrement the
'                       window counter variable.
'-----

SUB DitchWind SHARED
  WINDOW CLOSE WindEx
  WindEx = WindEx - 1
END SUB

```

PRODUCT INFORMATION

AC/BASIC

Absoft Corporation
2781 Bond Street
Rochester Hills, MI 48307
(313) 853-0050
\$125
Reader Service No. 386

GS BASIC

Apple Programmers
and Developers Assoc.
Apple Computer
20525 Mariani Ave.
Cupertino, CA 95014
(408) 996-1010
Reader Service No. 387

Micol Advanced BASIC

Micol Systems
9 Lynch Road
Willowdale, Ontario
M2J 2V6
(416) 495-6864
\$145
Reader Service No. 388

ORCA/BASIC

ByteWorks Inc.
4700 Irving Blvd. N.W.
Suite 207
Albuquerque, NM 87114
(505) 898-8183
\$95
Reader Service No. 389

TML BASIC

TML Systems
8837-B Goodbys
Executive Drive
Jacksonville, FL 32217
(904) 636-8592
\$125
Reader Service No. 390

something against which to weigh the design of PrintFile and TypeFile.

MsgDialog (msg\$,style%). Generates a standard dialog for your communication. Required parameters are a text string—the message you want to display—and the style of dialog. The procedure generates an "OK" button automatically. If style% has a value of 2, MsgDialog will also generate a "Cancel" button.

If you specify and click on the Cancel button, MsgDialog will set the value of the global variable cancel% to 1, as a way of communicating with the calling routine. You could use this value for a test, as in the following:

```

IF NOT cancel% THEN
  SortList
  ShowList
  CloseFiles
ELSE
  CloseFiles
END IF

```

DoWind. This procedure opens a full-screen document window with the screen dimensions established in DoSetUp (in the inCider.Shell). It also increments the global-window counter variable WindEx.

The reason for using global screen dimensions is portability. Screen size is one of the key differences between computers that support AC/BASIC-compatible compilers. By limiting hard-wired dimensions to one occurrence in the program, it becomes easier to adapt the program.

DoDWind. This procedure works like DoWind, but generates a window suitable for a dialog box.

DitchWind. This tool closes the most recently opened window, then decrements the window counter.

It's not the same old BASIC anymore—the GS challenges your creativity anew. From desk accessories to games, from education to commercial development, all the tools and support you need are within arm's reach. □

JOE ABERNATHY IS A PRODUCTION EDITOR AND OCCASIONAL ARTS CRITIC AT THE *HOUSTON CHRONICLE*. HE'S ALSO A CERTIFIED APPLE DEVELOPER AND THE AUTHOR OR COAUTHOR OF EIGHT APPLE II PROGRAMS. WRITE TO HIM AT P.O. BOX 66046, HOUSTON, TX 77266-6046.