LEARNING FUTUREBASIC

MACINTOSH BASIC POWER

By L. Frank Turovich

SENTIENT FRUIT

LEARNING FUTUREBASIC™

Macintosh BASIC Power

by L. Frank Turovich

Copyright ©1994 Sentient Fruit

All Rights Reserved Worldwide

Requirements

The source code in this book was designed to work with the FutureBASIC™ programming language. All development was done under FutureBASIC version1.02.

Learning FutureBASIC: Macintosh BASIC Power

by L. Frank Turovich

Questions

Zedcor, Inc. does not support, nor is Zedcor, Inc. responsible for any of the contents of this product. All inquiries and questions regarding this product or any other products produced by Sentient Fruit should be directed to...

Sentient Fruit ™

MACINTOSH CONSULTING • PROGRAMMING • DOCUMENTATION

PO BOX 13362 • TUCSON • AZ 85732-3362

We can also be reached online at:

America Online: TUROVICH

Internet: turovich@aol.com

Print history

Copyright ©1994 Sentient Fruit

Second Printing March 1994

Printed in the United States of America

ISBN 0-9639552-0-9

Trademarks

FutureBASIC is a trademark of Zedcor, Inc.

Macintosh and ResEdit are trademarks of Apple Computer, Inc.

Sentient Fruit and the Mac-in-the-Tree logo are registered trademarks of L. Frank Turovich.

All other products and logos mentioned in this documentation are the trademarks or registered trademarks of their respective owners.

Legalese

LIMITATION ON WARRANTIES AND LIABILITY

EVEN THOUGH SENTIENT FRUIT AND L. FRANK TUROVICH HAVE REVIEWED THIS MANUAL, SENTIENT FRUIT AND L. FRANK TUROVICH MAKE NO WARRANTY OR REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS" AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY. IN NO EFFECT WILL SENTIENT FRUIT OR L. FRANK TUROVICH BE HELD LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACCY IN THIS MANUAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL, OR WRITTEN, EXPRESS OR IMPLIED. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

Production

This book was written on a Macintosh II, System 7.1 using FrameMaker® for the words, FutureBASIC™ for the source code, Resorcerer® and ResEdit™ for the resources, DeskPaint®, DeskDraw®, and ClarisWorks™ for the graphics. It was printed on a Texas Instruments microLaser PS-35 at 300 DPI using Bookman for body text, Helvetica for titles, Courier for source code, and Symbol for bullets.

Acknowledgements

This book is the cumulation of months of hard work both in writing and programming. It could not have happened without the support of the following people:

First and foremost, I must thank my loving wife Lora for her continued support and great patience in editing numerous revisions.

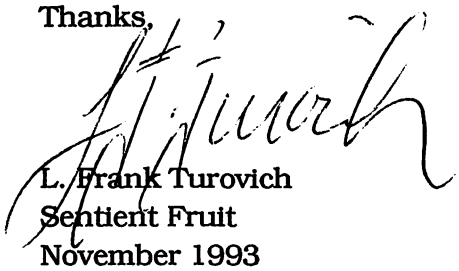
John "the pitbull" Richetta for his ruthless editing style. All mistakes remaining herein are mine alone.

The brothers Gariepy: Michael, the CEO of Zedcor, Inc. for allowing me to do it, Peter for the bully cover design, and Andy for answering my endless questions.

Ross Lambert of Ariel Publishing for suggesting it.

And finally to my test readers who saw a very early version of what you now hold in your hands: Chris Stasny, Chris Dwyer, and Paul Valach.

Thanks,



L. Frank Turovich
Sentient Fruit
November 1993

Table of Contents

CHAPTER 1

Introduction

Welcome

Welcome to *Learning FutureBASIC™: Macintosh BASIC Power.*

Since its release *FutureBASIC* has rightly gained a good share of enthusiastic followers. Its ease-of-use, extensive BASIC command set, built-in access to the Toolbox, flexible environment, and versatility in creating both stand-alone applications and code resources have been welcomed with open arms by the BASIC community. *FutureBASIC* has made the writing of programs easy and fun for professionals and hobbyists alike.

Old followers of *ZBASIC™* (*FutureBASIC's* predecessor) are amazed at the wealth of new features contained in this powerful language. New followers find its arsenal of capabilities astounding. Professionals who haven't used BASIC in years are finding it easy to write code in hours instead of days.

Worldwide, people are using it to create programs that range from CASE tools to Mac-to-mainframe communications, from QuickTime™ editors to educational software, from graphics programs to children's games, and everything in between.

The Rise of BASIC

BASIC has always been a popular language. Since its beginning in the early 1960's, BASIC has been available on nearly every computer ever created. Its inclusion with most home computers guaranteed a wide distribution, and its ease of learning made it the language most beginners turned to when they started to program.

Sadly, BASIC has always had a reputation for being too slow (most BASICs were interpreted), too general (it has commands for nearly everything), and too clumsy when it came to writing "real programs".

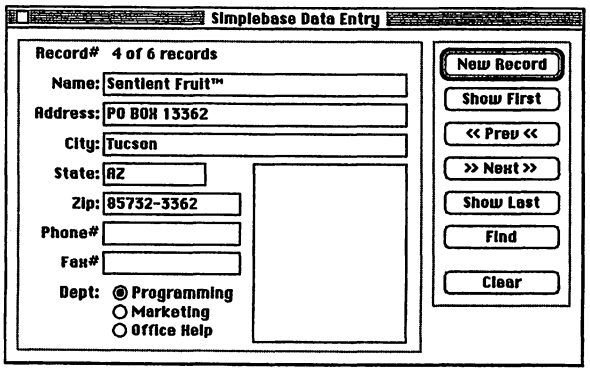
But *FutureBASIC* (hereafter referred to as FB) addresses all of those points and more. FB is a compiler, so it's fast. While it does have an extensive 300+ command set, it doesn't try to do everything, just enough to let you get the job done. Nor is it clumsy: FB offers all the high-level structures found in other programming languages including SELECT CASE, LOCAL FNs, INCLUDE files, access to system equates, and much more. With FB you can write "real programs" quickly and easily, and you can create all of those widgets that make the Macintosh so much fun. For example, you can write desk accessories, control devices, custom windows, menus, and controls – the choices are endless.

The Plan

The outline for this book is simple. Users have been demanding a comprehensive book that would lead them step-by-step into the exciting world of Macintosh programming. While the FB manuals have all the necessary clues, those clues aren't really drawn together into a complete, full-featured application where a new user can get all the answers. The *Getting Started* manual included with FB nibbles around the edges, but never really tackles the tough issues facing a beginning Mac programmer.

With this book, I've tried to anticipate and respond to your questions as they might come up. I walk you step-by-step through an entire Macintosh application, from concept to completion. Along the way, I'll also explain what goals we want to achieve, how we'll accomplish those goals, as well as providing helpful insights into the reasoning behind these choices.

FIGURE 1. Data Entry window in SimpleBase.



SimpleBase

To provide a reference point for explanations, I wanted an example program that would draw on a host of Macintosh features, as well as using Toolbox commands and resources. What I came up with is *SimpleBase*, a small database program that a business might use to track employee information.

As I said, it's simple, so I won't be going into how to calculate employee benefits or taxes, but I will examine all of the common Macintosh features we've all come to know and love. Things like menus, windows, buttons, scroll bars, alerts, printing and much more will be examined in depth. Each will be introduced as it is required by the program along with copious explanations along the way.

I've also attempted to throw in just about every technique, shortcut, or obscure bit of knowledge I've acquired through years of programming the Mac – information I would have killed for when I first began. Most of these never appeared in any programming book, but were discovered after lots of sweating, cursing, rebooting, and in conversations with fellow programmers. Of course, pure inspiration also played a big role.

Theme of Things

With all this explaining to do, I decided I needed a nifty theme to run through the book, one that really ties everything together and keeps people interested. I chose to use an exercise theme as my guide, since it seemed to fit both the style of this book as well as my current workout regiment. Thus each chapter is divided into the following four sections:

Warm-up

Each chapter starts by telling you what the chapter offers. It's followed by a few terminology definitions, explanations of various common programming tasks relevant to the chapter topic, as well as numerous step-by-step program examples, insightful illustrations, and lots of straight talk about why you should adhere to Apple's interface standard. You'll get your feet wet on some fundamental Macintosh programming ideas, but the water will never rise over your ears.

Regular Exercise

This section is where we'll actually develop the program code for the *SimpleBase* application. I'll walk you step-by-step through the various routines explaining their purpose, as well as showing you their development from initial concept to final implementation.

Each chapter in the project has one or more files on disk that show exactly how each program step progressed. Print them out, pull up a lounge chair under your favorite shady tree, sip some ice tea (no sugar please), and examine them as you read each chapter in order to better understand the project.

Peak Performance

This is where we pull out all the stops. You may want to skip this section the first time through as it is more complex and isn't always relevant to the chapter topic. However, once you've completed the project, come back and browse through it for advanced tips and programming techniques that you can use to modify *SimpleBase* or your other programs.

Cooldown

Finally, we wrap up all the concepts explained in the chapter into a neat little bundle. This will emphasize the chapter's main points, and point out a few key concepts you should remember.

Where Should I Start?

Like they always say, "an application of 10,000 lines begins with one keyword," or something like that. That's exactly how you should approach this book – one step at a time.

Never Programmed Before?

If you don't already have it, call Zedcor at 602.881.8101 and order their "Programming the Macintosh with FutureBASIC" manual. This interactive manual takes you from programming ground zero all the way to handling simple Macintosh features. A good place to start if you haven't done any BASIC programming before.

Head down to your local library and pick up a few programming books that explain BASIC programming concepts. Explore the possibilities in those books until you feel comfortable with loops, data types, and structured programming concepts.

Note that most of these books are not geared to programming the Macintosh, but are aimed instead at other computers. Believe me, they're still useful. You'll probably have to fiddle with some syntax differences, but with a few changes, many of their exercises will run just fine.

When you feel ready, start at the "Program Design" chapter, concentrate on the Warm-up section to get the main concepts behind each chapter. Then, go through the Regular Exercise section carefully as we create the *SimpleBase* project, making sure you understand each step before going on to the next.

When you've finished the project, use the same techniques explored here to create your own program. Say to yourself, "Okay, for my program I need to add...", and then review the appropriate chapter to add a particular program feature. Finally, go back and read the Peak Performance sections to learn even more.

If You've Programmed BASIC Before

Read through the Warm-up sections to get the main concepts for the chapter, then read the Regular Exercise section and create the *SimpleBase* project. Go back after you've finished the project and reread the Peak Performance sections for additional programming knowledge you may find useful in other projects.

References

If you don't know it already, I'll tell you now: you can't program in a vacuum. Programming the Macintosh requires knowledge of the entire Macintosh, from interface to the nitty-gritty of byte passing. Here are some references I have found very useful, both in writing this book, and in my daily programming endeavors.

Inside Macintosh, 2nd Edition

The absolute best reference work for programming the Macintosh is Apple's own *Inside Macintosh* series. Even as I write this, the second edition series is almost completely published. The 18 volumes replacing the original six have expanded both in size and depth of explanation. If it's about programming the Macintosh, you'll find it there.

If you do any type of commercial or consultant programming, you shouldn't be without any of them. If you're a weekend programmer, I urge you to get the volumes that most interest you. In either case, you won't be sorry.

Macintosh Revealed

The second reference work I highly recommend is the *Macintosh Revealed* series by Stephen Chernicoff. While all of his examples are written in MPW Pascal, the explanations are clear, and the progression from feature to feature is interesting and concise. If you can't get all of the *Inside Macintosh* series, try this four book set for an abbreviated reference work.

Inside Basic

Of course, for the most up-to-date information on *FutureBASIC* and how you can make it work, read *Inside Basic* magazine from Ariel Publishing. Each monthly issue contains several articles that provide numerous hints, tips, explanations, and answers to common *FutureBASIC* programming questions.

America Online

For daily conversation with hundreds of *FutureBASIC* users, you can't beat the forums on *America Online*. Both Zedcor (keyword: ZEDCOR) and Ariel (keyword: ARIEL) provide *FutureBASIC* support areas where you can ask for and receive help, usually in hours. If you've got a modem, get signed up and join the fun, you won't be disappointed.

Internet

There is also a list on the Internet where *FutureBASIC* owners can keep in touch and up to date. The addresses to become part of the list include:

Add name:	`futurebasic.list-request@statistik.tu-muenchen.de`
Send message:	`futurebasic.list@statistik.tu-muenchen.de`
List owner:	`futurebasic.list-owner@statistik.tu-muenchen.de`

Sentient Fruit

Finally, you can contact us here at:

Sentient Fruit
P.O. Box 13362
Tucson, AZ 85732-3362

Or contact us online at:

America Online:	TUROVICH
Internet:	`turovich@aol.com`

Conventions

The following is a list of formatting and presentation conventions used throughout this book.

Typographical Info

Program listings and examples, as well as routine and variable names, all appear in `Courier`. Small program examples will appear in-line with the main body text like this:

```
PRINT%(10,10) tmp$
```

Larger examples will appear offset from the main text and be referenced using the `PROGRAM` identifier.

FutureBASIC keywords appear in the text as uppercase `Courier` font like this: `WINDOW(_efNum)`.

Toolbox keywords in the text appear in mixed case `Courier` like this: `SetRect`.

Menu titles, item names, and buttons always appear in **Chicago**.

Program names appear as *italic*.

Key terms and concepts are always shown in boldface the first time they appear, like this: **keyword**.

Since we will use `LOCAL FN`s for all of our programming examples they will be referred to collectively as functions, routines, and subroutines for variety.

Occasionally, a program line will extend past the right of the page and onto the next line. Such lines are marked with a "¬" symbol to indicate this continuation.

Notes that explain a particularly obscure point will appear within the body text as smaller, offset type, preceded by a small dividing line. Read them for additional background material.

- *A note explains something not totally relevant to the main discussion, but interesting just the same.*

The entire *SimpleBase* program appears in the Appendix.

A bibliography of related books and magazines has also been provided in the back of the book.

Let's Get Started...

Now that we've gone through all the uninteresting, but necessary, preliminary information, let's begin learning how to program the Macintosh using *FutureBASIC*.

CHAPTER 2

Program Design

Warm-up

This chapter introduces structured programming. Along the way we'll learn:

- Top-down design methodology,
- Stepwise refinement techniques,
- The only three control blocks you'll ever need, and
- How to organize your program layout.

Additionally, we'll describe design methods that make writing programs:

- easier to write correctly the first time (fewer nasty bugs),
- easier to read (self-documenting),
- easier to modify, and
- easier to take apart and re-use.

This type of program is much easier to work with than those created by people who fail to understand or practice these methods.

At the moment, this may seem boring or a bit overwhelming. However, as you grow in programming experience, you'll realize that the early application of these programming techniques will pay back dividends in increased productivity, more error-free programs, and more re-usable code.

Programs

A **program** is a sequence of instructions that describe how the computer is to perform a defined task or goal. It doesn't matter what the task or goal is since a computer doesn't know, doesn't care, and couldn't be bothered with anything but executing its instructions. A **programmer** is an individual that writes the necessary step-by-step instructions that tell the computer how to accomplish the task or achieve the desired goal.

Most programs have a major task or goal to accomplish. A word processor program handles tasks related to writing. To accomplish this task, hundreds, if not thousands of sub-tasks exist that enable the program to open a file, edit it, and save or print out the updated file.

To accomplish all its various tasks, a program contains a hierarchy of subroutines. A **subroutine** performs a specific task for the program. In the word processing example, some subroutines allow the user to create a letter, others allow the letter to be saved on disk, yet others allow the user to print the letter.

By calling different subroutines in a particular order, the program can perform all of those actions and more. Thus, a program consists of a series of different subroutines, each of which perform a single act, linked together in a specific order to perform a particular action. The programmer determines the series of subroutines that will perform the task, writes the routines to complete the task, calls them in the correct order, then tests them separately and together to ensure that they accomplish the stated goal.

When writing a program, a programmer seldom gets it right the first time. Errors in design, faults in logic, incorrect parameters, and many other oversights all contribute to what are commonly called program **bugs**. Bugs are the downside of the learning process. The process of finding and eradicating program bugs is known as **debugging**.

Top-Down Design

The method of program design used throughout this book is called **top-down design**. Top-down design has the programmer think about the program in its broadest design terms, then slowly refine the design step-by-step down to its exact details.

Top-down design encourages programmers to think in abstract terms about the actual goal or task the program is supposed to solve. Once a goal is identified (i.e. printing a letter), sub-goals to print the letter are defined. The letter must be open, it may have to be formatted, the printer must be readied, the letter must be sent to the printer, and finally, everything cleaned up afterward. All of these are sub-goals that must be accomplished successfully in sequence to print the letter. Each sub-goal can be used to identify a program subroutine. The subroutine must then be examined and its sub-goals also determined. It's time to write code when no additional sub-goals can be identified.

Along with top-down design, we'll also use another technique known as **stepwise refinement**. Stepwise refinement is based on the top-down design model where a program goal is defined, then code is written that will achieve the goal in greater and greater detail, hiding the details of implementation until the very last routine.

Stepwise refinement allows us to test concepts of the design at each step along the way using **skeleton** routines. Skeleton routines are subroutines that do nothing, yet allow us to test for operational correctness at the level of detail we are working on. Most will contain a single `PRINT` statement defining its task, like "Print routine called" or something similar. By executing the code at each step we assure ourselves that the routines on that level of refinement operate correctly. Once we're satisfied that everything works properly, we move down a level and begin filling in the details at that level.

Let's look at the printing example. We start by defining the main task as a subroutine in a `LOCAL FN` structure like this:

```
LOCAL FN PrintLetter
END FN
```

Next, we refine the steps to actually print a letter using the sub-goals previously mentioned. The print routine now reads like this:

```
LOCAL FN PrintLetter
  FN OpenLetter
  FN EnsurePrinterIsReady
  FN FormatLetter
  FN SendLetterToPrinter
  FN FinishLetter
END FN
```

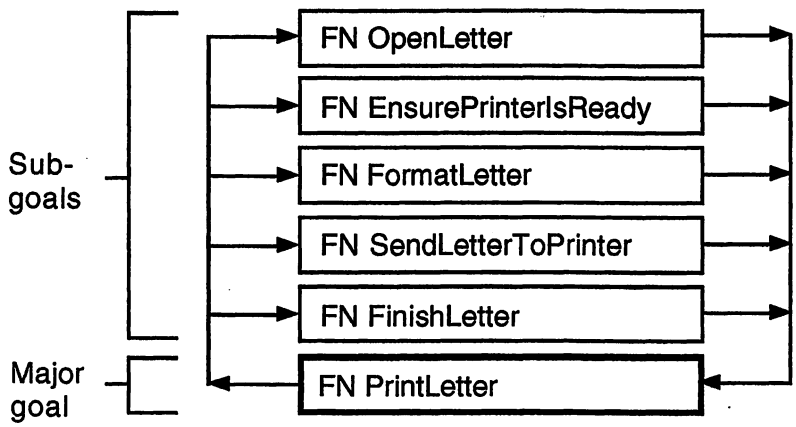
Each sub-goal has its own subroutine. The details of opening, formatting, and printing a letter are hidden in the various subroutines as shown in Figure 3. This hiding of details allows us to concentrate on the goal, and not be distracted by details. However, once this routine is defined, its time to delve into the various subroutines one at a time and implement their tasks. Each of these subroutines can also call other, even lower level subroutines to accomplish their individual tasks.

Note that we can test the `PrintLetter` subroutine at any time to ensure that it works correctly. Once the `OpenLetter` function works, we can implement the `EnsurePrinterIsReady` function. Each function is independent of the others. This allows us to test them individually and, when all are complete, test them together to assure ourselves that they work properly.

The use of these techniques offers several benefits including:

- At each stage of the top-down process you're concerned with the details of how the program operates only at that level. Once you're satisfied that it operates correctly, you move down a step and refine the next level. Repeat until all details of the design have been fleshed out completely.

FIGURE 2. Major goal with sub-goals.



- This technique of chunking the program into smaller, more easily managed sections enables you to ignore the other portions of the program until required. A large program is simply several smaller programs linked together, not a large single set of program instructions.

By now, some of you are probably thinking this is a total waste of time. "Why? I could go in and code something up in no time at all," you're thinking, "Why worry about levels and steps?"

Well, that may be true for small programs of 100-200 lines. But, when it comes time to code a far larger program, say 5000 to 20,000 (or even more) lines of code, this attention to design, implementation, and detail at the upper levels will greatly improve your ability to complete the project faster and with fewer bugs.

Greet User by Name Example

Eventually, after all your top-down design work, it comes time to begin writing the code to accomplish the stated goal. One way to think about implementing this code is in **pseudocode** format. Pseudocode is simply a series of statements that describe the actions the subroutine will perform in English, or whatever language you use daily. For example, the pseudocode for our program is shown in Program 1.

PROGRAM 1. Sample program pseudocode.

```
1. ask for user last name
2. ask for user first name
3. combine the names with a salutation
4. show the result to the user
```

Note that there are no FB keywords included in this pseudocode example. We'll do that later. For now, it's important to see how to describe to ourselves what the computer should do. This allows us to focus on the program steps required to tell the computer how to perform the task.

- *Note that the line numbers are merely used to reference a particular line in the text. They will not be used in the final program at all.*

Converting this pseudocode to FB keywords is pretty easy to do. Asking for the user's last and first names in lines 1 and 2 tells us to use a statement that provides for such user interaction. In this case, the `INPUT` statement will serve handily. We combine the two string entries in line 3, then display the final result using a `PRINT` statement. Program 2 shows how we converted the pseudocode into source code with keywords that achieve the desired task.

PROGRAM 2. Sample code first translation.

```
INPUT "What is your last name? ";lastName$
INPUT "What is your first name? ";firstName$
salutation$ = "Hello, "+firstName$+" "+lastName$+"."
PRINT salutation$
```

As you can see from this example, it's entirely possible to translate pseudocode directly into statements that accomplish a stated program goal. This was an extremely simple program, yet it displays the same characteristics of a large program. Any task can be broken down into individual pseudocode steps that can themselves be translated into FB commands. Later, as we develop *SimpleBase* you'll see how a single line of pseudocode can lead to multiple subroutines.

Why ask Why?

Good top-down design will enable you to design and write programs faster and with less chance of error. Because you test at each level of the design process using skeleton routines, errors are kept to a minimum.

The steps involved in good top-down designing are:

- Begin with a simple main idea. Write in pseudocode the major tasks to accomplish.
- Translate each step into detailed pseudocode that resembles one or more BASIC statements.
- Write skeleton routines to test your design at each step of the process.
- Fill in subroutine details as required. If more subroutines are required, repeat the above steps for each level.
- When the program is working correctly, implement any additional improvements as desired.

Control Structures

Earlier we mentioned that a program consists of many individual subroutines linked together to perform an action. If the programmer could write those instructions as a series of linear statements, (i.e. one line right after another in sequence) programming would be extremely easy to do. However, a linear program is nearly useless. It can't branch to another subroutine, it can't loop upon itself, and it can't handle the real life problems it's supposed to solve.

Since a program consists of simple statements like PRINT, GOSUB, and INKEY$, where do programming problems come from? Each statement is simple to understand, yet when combined with others they can quickly achieve great complexity. Even the short example shown in Program 3 demonstrates how a few lines of code can cause confusion in a program.

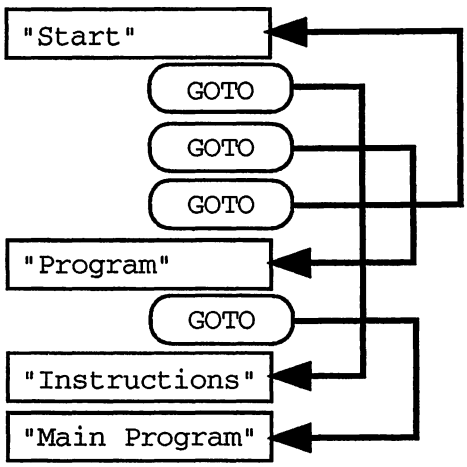
PROGRAM 3. Complex statement example.

```
"Start"
INPUT "Do you want instructions? ";answer$
IF answer$ = "YES" THEN GOTO "Instructions"
IF answer$ = "NO" THEN GOTO "Program"
PRINT "Please enter YES or NO only."
GOTO "Start"
"Program"
GOTO "Main Program"
"Instructions"
PRINT "Here's your instructions..."
PRINT "..."
"Main Program"
and so on...
```

Do you understand what Program 3 is trying to do? You probably had to study it awhile to be entirely sure. Its goal is to find out if the user wishes to view the program instructions, and then display or skip them at the user response. If it's anything but YES or NO, the loop repeats. Pretty straightforward and easy to follow, not!

A count of branches show that there are four input points (the subroutine labels) and four exit points (the four GOTO statements) in this 12 line program. Each branch implies a different point of entry and exit. The difficulty comes in trying to understand which branch occurs and under what conditions. A diagram of these various jumps looks something like the spaghetti shown in Figure 3.

FIGURE 3. Diagram of confusing code.



How even simple looking code can go bad using improper programming techniques.

Is this how you program?

Can you follow it? You probably had to trace through the code several times to figure out where all the branches were taking you and under what conditions. If this much confusion can creep into only a dozen lines of code, imagine this type of decision making over hundreds or thousands of program lines.

With this type of branching, could you ever be absolutely sure that it works correctly under all conditions? Would you want to debug or try to maintain this type of code? Probably not. So let's examine a way to avoid this kind of confusion.

Linear Programs

There is a solution: make every program a linear program. But wait, we've already said that's not possible. Well, sure it is, if we bend the rules slightly. Instead of treating each statement separately, let's treat them as a larger block of statements. These block statements can be combined into a linear program even though the individual statements contained within them aren't linear at all.

Linearity is now imposed upon the block statements. Each block must completely finish executing all of its individual statements before another block can execute. So when a block is called, entry is always at the top of the block and exit is always at the end. This form of block is often called a **one-in/one-out** block. Any jumps out of the block must return to that block in order to uphold this one-in/one-out sequence.

- *In this we follow the Pascal convention. Pascal doesn't allow one to exit any structure from the middle, while C doesn't care where one exits. It's akin to the old* GOTO *statement which originally led to the spaghetti code syndrome of illogical jumps and unreadable programs.*

We'll enforce the one-in/one-out block structures in *SimpleBase* by using LOCAL FNs to define every subroutine in the program. If you're used to a more free-wheeling programming style this method may seem a bit restrictive at first. "Only one entry and exit point? It'll never work," you say. Believe me, after using it awhile and seeing how it simplifies your programming task, it'll soon become second nature and you'll never go back.

- *The latest version of FutureBASIC includes an* EXIT FN *statement to satisfy those people who thrive on confusion in their code. We won't use it here and I don't recommend that you use it either.*

Okay, now that we understand the linear programming method, let's look at three types of control blocks that will help us to achieve a linear programming style.

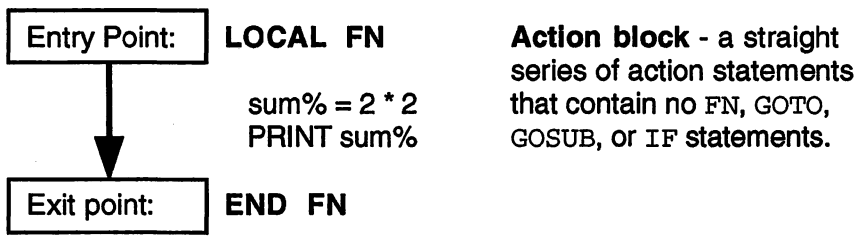
Action Blocks

The most rudimentary control structure is the **action block**. It's easy to overlook because the action block is simply a sequence of action statements.

Identifying an action block is easy: it can contain no IF, GOTO, GOSUB, or FN calls or other control statements that might cause a jump outside of the block. An action block might look like the example shown in Program 4.

This series of statements direct text output to a printer and then returns it to the screen. It contains no branching or alternate actions; it is a perfect action block example.

FIGURE 4. Action block structure.



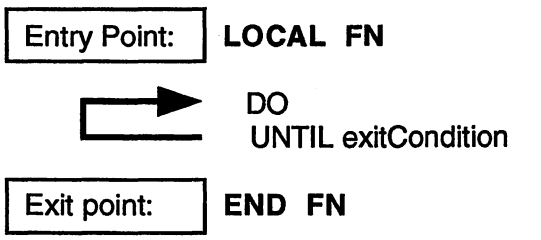
PROGRAM 4. Action block example.

```
LOCAL FN PrintSomething
  ROUTE _toPrinter
  PRINT "Hello there!"
  PRINT "What a wonderful day it is."
  ROUTE _toScreen
END FN
```

Loop Blocks

The next control block structure is called a **loop block**. As you might suspect, it is used to repeat a specific action several times. Loop blocks are always entered at the top of the control structure and exited at the ending control structure when the exit condition becomes true. This is graphically shown in Figure 5. There are several types of loop structures available including FOR/NEXT, WHILE/WEND, and DO/UNTIL. The example in Program 5 demonstrates a standard loop block construction.

FIGURE 5. Loop block structure.



Loop block - a loop structure that continuously executes until the exit condition is true.

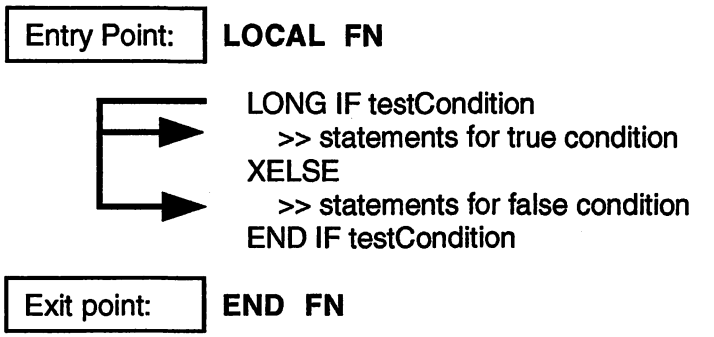
Just because it's a loop block doesn't mean you can't branch out of the loop. The key is to always return to the loop and then exit. FN and GOSUB statements work perfectly for this since they automatically return to the line following their call. Never use the uncontrolled GOTO to exit a loop block. It's not only bad practice, but causes more problems than it's worth.

- *Many people believe they should be allowed to exit in the middle of a loop structure, a la C. But the one-in/one-out rule must hold for loops also.*

PROGRAM 5. Loop block example

```
count = 0
DO
  PRINT "I now equal = ";count
  INC (count)
UNTIL count > 100
```

FIGURE 6. A branch block structure.



Branch Blocks

Programs would be worthless if they performed identically from one run to another. **Branch blocks** allow programs to execute different code depending on the condition of one or more variables.

Branch blocks direct program control to other portions of the program. This is graphically illustrated in Figure 6. Once we've entered a branching block at its entry point, we must exit at its end, never in the middle. There are several branching structures available including `IF/ELSE`, `LONG IF/XELSE/END IF`, and `SELECT/END SELECT`. A good example of a branching block is shown in Program 6.

PROGRAM 6. Branch block example.

```
LOCAL FN GoSomeWhere (direction)
  LONG IF direction
    FN HandleTrueCondition
  XELSE
    FN HandleFalseCondition
  END IF
END FN
```

Nesting Block Structures

In 1966, Boehm and Jacopini mathematically proved the following theorem:

> Any program logic, no matter how complex, can be resolved into action blocks, loop blocks, and branch blocks.

If this seems a bit extreme, note that we can nest one block structure inside another. Thus, the "do something" statements inside a loop or branch block can consist of another action, loop, or branch block.

This nesting of blocks ensures that our entire program follows the one-in/one-out principle. Instead of a straight sequence of program statements, a program will consist of a sequence of action, loop, and branch blocks.

Referring back to the ill-written "ask instructions" example in Program 3, we can rewrite the code to take advantage of our new knowledge of the one-in/one-out format using all three control block structures. The rewritten code can be seen in Program 7 with some flow arrows pointing out the program direction. When completely diagramed out as shown in Figure 7, we can see that complexity has been reduced to a single loop branch until the user responds correctly.

The new code begins with an action block at `"Start"`, then immediately switches to a loop block. While in the loop, we examine the condition of the `doneflag%` variable to determine when to exit the loop. When `doneFlag%` becomes true, a yes or no answer has been received, the loop completes and the action block continues. Next, a decision block is used to check the status of `answer$`. If `answer$ = YES` we display instructions, otherwise it skips the entire `"Instructions"` subroutine. In either case, the action block continues with the `"Main Program"`, and so on. As you can see, only one entry point (`"Start"`) and one exit point (the end of the program) exist in the program.

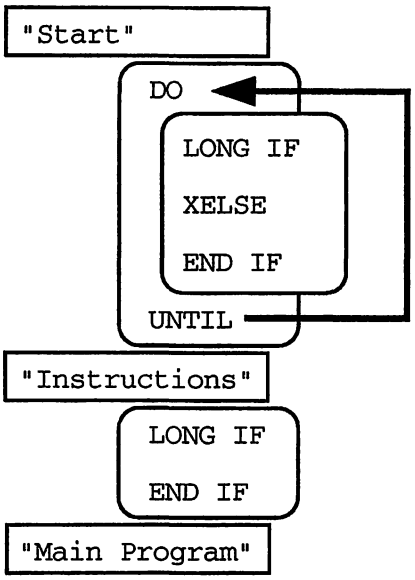
Don't you think this code is much easier to read and follow than the original? There are no illogical jumps, no multiple entry or exit points. This lessening of complexity makes writing large programs infinitely easier to design, write, and debug.

PROGRAM 7. Improved complex statement example.

```
"Start"
doneFlag% = _false
DO
   INPUT "Do you want instructions? ";answer$
   LONG IF answer$ = "YES" OR answer$ = "NO"
      doneFlag% = _true
   XELSE
      PRINT "Please enter YES or NO only."
   END IF
UNTIL doneFlag%

"Instructions"
LONG IF answer$ = "YES"
   PRINT "Here are your instructions..."
   PRINT "..."
END IF
```

FIGURE 7. Optimized instructions asking routine.



An optimized linear program using action, loop, and decision blocks.

Wouldn't you like your code to read like this?

Program Layout

"A place for everything and everything in its place." That's the nature of our program layout. It formats a program listing in a way that makes it easy to find specific program features. It also makes it easier to navigate and change portions of your program because like routines are grouped together. The following are descriptions of the various sections used in creating *SimpleBase*. A sample of this program layout can be found on disk as *00.SimpleBase.main*.

Header Section

Use the Header section to define all the compiler directives for accessing resource files, setting compiler options, defining output files, and anything else that doesn't have a particular place. I usually have a header that looks something like this:

```
RESOURCES "resource file.rsrc", "APPL9999"
COMPILE 0, _strResource _macsBugLabels
OUTPUT FILE "my application name"
```

RESOURCES specifies the resource file to use with the program, the program's final type (always APPL for applications) and creator (any 4-character sequence not already in use). The COMPILE statement tells the runtime to convert all the strings used in the program to STR# resources when the program is built, as well as adding labels that the *MacsBug* debugger can

display, should a program bug surface. Finally, I use the OUTPUT FILE statement to specify a default filename for building the final application. It prevents me from having to enter a name each time I build the program.

Constants Section

Here is where we define all of our program constants. **Constants** are integer and long integer values represented by a name. Constants are identified by a leading underscore character. When FB compiles the program, it replaces each constant name with the actual value of the constant. You can update an entire program by changing a single constant value.

There are three types of constant definitions used: *FutureBASIC*, user-defined, and Macintosh constants. Let's look at each type in turn.

FutureBASIC Constants

There are over 100 pre-defined *FutureBASIC* constants that provide mnemonic support for FB statements and functions. A quick examination of the FB Reference manual reveals that many statements and functions already have common constant names associated with them. For example, instead of remembering which number represents which button type, it's easier to remember: _push, _checkBox, _radio, and _shadow.

Yes, it may seem a drag at the time to enter a window type of _doc instead of the number 1. Why use four letters when I could type only one instead? Well, months (if not years) later when you reread the source code, it's much easier to understand which window type is built by name rather than by number.

User-defined Constants

User-defined constants are constants that you design for your own program. They can represent any integer or long integer value you require. You define a constant like this:

```
_myConstant = myValue
```

Where _myConstant is the constant name and myValue is the actual value. Each constant must be defined on its own program line and must be unique in the first 16 characters.

You will see in later chapters how we often create user-defined constants to represent common values used throughout the *SimpleBase* program. This usage helps to make the code almost self-documenting. For example, instead of representing the main Data Entry window like this:

```
WINDOW #1,"DATA",(0,0)-(500,300)
```

We will instead write it as:

```
WINDOW #_dbEntryWIND,"DATA",(0,0)-(500,300)
```

Now, just a glance at the line will tell us which window it's creating. No more guessing. This also makes it much easier to read decision making code like this:

```
LONG IF wndID% = _dbEntryWIND
   ' do something useful
END IF
```

While you might remember the window number while working week after week on your killer mega-program, several months after you finish, it will read like total gibberish as you struggle to remember which value represents which window.

Macintosh Constants

In addition to FB constants, there are literally thousands of other predefined constants available for use. There are constants defined in the *Inside Macintosh* volumes (called equates), which the Toolbox managers like Window, Text Edit, and Dialog manager, as well as thousands more use all the time.

With these definitions, we can use them too. For example, four of the constants associated with a rectangle record are: `_top`, `_left`, `_bottom`, and `_right`.

Globals Section

The Globals section of a program contains all variable definitions determined to be global in nature. That means that all subroutines and functions (including `LOCAL FN`s) have access to their values without explicitly passing them as parameters.

Globals are normally kept in one or more global files, separate from the main program. They are called by the main program using the `GLOBALS` statement with the file name. This separation of global settings from the main file allows other files, namely include files, to also make use of the same global definitions.

Global variables should be kept to a minimum. Doing so reduces the chances for variable conflicts, and promotes portable code since self-contained subroutines are easily copied to other programs and can operate without modification.

Functions Section

The Functions section of a program contains all of the subroutines required to execute the program.

Functions can also be contained in files external to the main source file. These external files are called **include** files. We'll see in later chapters how you can write common routines once and put them in an include file for use by many different programs.

Main Loop Section

The Main Loop section is where all event handling takes place. Only a single one of these is allowed in a Macintosh program. Other BASICs promoted the bad practice of multiple event handling loops in their examples. This is not correct, nor an encouraged practice in a Macintosh program. All events are captured in a single event loop, then directed to subroutines designed to handle the event.

As we'll see later, the Macintosh operating system sends a program messages. The Main Loop section is where the program receives those messages and directs each of them to the appropriate subroutine.

SimpleBase Notes

When entering the *SimpleBase* program it's important to keep in mind a few rules:

- A function must be defined before it can be called in a subroutine. Example: `FN One` must be defined before `FN Two` can call it.

```
LOCAL FN One <-- defined before FN B
END FN
LOCAL FN Two
   FN One
END FN
```

- A function does not have to return a value. It can act as an action block and just do something.
- You can't have two functions with identical names. Always enter the example code into the specified subroutine. Some subroutines undergo massive changes during program development so be careful when appending new code to an already created subroutine.
- *Learning FutureBASIC: Macintosh BASIC Power* assumes you are using the default preference settings FB shipped with. If you are using the "Programming the Macintosh with FutureBASIC" workbook, reset your preferences back to the default before continuing with this book.
- As we create *SimpleBase*, all of the interim files can be found on disk. To see exactly how the program was built for a particular chapter, just go to the folder with the chapter's name. Inside are all of the source code files, numbered sequentially, to illustrate the order in which subroutines were added or updated.
- If you create the program along with the book, note that you will normally create an initial subroutine, then add additional code to it as we continue to develop *SimpleBase*. Do not simple create a new subroutine with an identical name.

The complete *SimpleBase* program is contained in the Appendix at the back of this book. Use it as a reference to locate subroutines in your version of *SimpleBase* while following through this book.

Cooldown

Whew! That was a lot of information for a beginning chapter. Along the way we learned about structured programming techniques called top-down design and stepwise refinement. We also saw the importance of using these techniques to help us design and write programs faster, with less bugs, and which are much easier to maintain.

We also talked about the three control block structures that help us write linear programs. We saw how we can use action, loop, and branch blocks to control program flow in a one-in/one-out sequence that makes for cleaner program designs. We also saw how we can nest these into a linear design that made it easy to follow a program's logic.

Finally, we talked about the layout of programs and described the various program sections we use to create the *SimpleBase* project.

Now we're ready to begin with our program, so let's get started.

Events

Warm-up

This chapter introduces events, the single most fundamental topic to understanding how to program the Macintosh. This topic is so pervasive that one chapter can't hold it all and will be distributed out over the entire book as required. In this chapter you will learn:

- What events are,
- The types of events, and
- How to program for events.

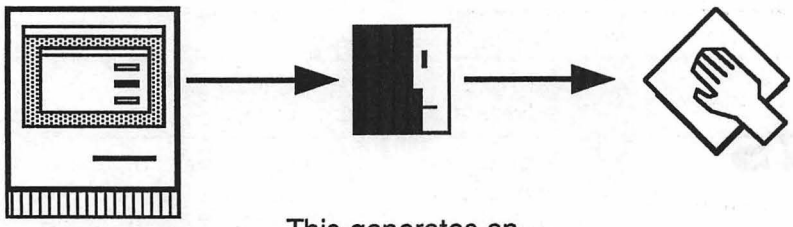
Once you've mastered this topic, you'll be well on your way to becoming a successful Macintosh programmer.

What are Events?

Events are messages from the Macintosh operating system to a program. Events inform programs that the user has selected a menu item, clicked the mouse, pressed a key, inserted a disk into a floppy drive, clicked a button, or performed some other noteworthy action. Events from the operating system tell an application to move to the background, come forward, refresh a window, update, and other. Yet other events enable programs to communicate with each other and transfer information between them.

Events are passed to the program via an **event queue**. The event queue is a first-in, first-out buffer (FIFO) area where events are lined up one after another as they are detected by the system. Up to 64 events can be in the

FIGURE 8. How events are generated.



User presses a button on the screen.

This generates an event that is seen by the operating system. It places the event into the event queue for the program.

The program examines the event and responds.

queue at one time. When a new event is generated that cause the queue to exceed 64, the oldest event is lost as the new event is placed in the queue.

Alright, why should you care about events?

On a Macintosh the user chooses how to interact with the program. A user can select a menu item, click a button, enter text, move windows, draw shapes, and perform a multitude of other tasks. All of these actions generate events. And, when the program receives an event, it should respond appropriately. In other words, a program waits until the user makes a choice, then promptly executes the necessary subroutines required to fulfill the request. The simple truth to successfully using events can be summed up by saying:

Never anticipate an event!

When you anticipate an event, you are actually fighting against the very nature of the Macintosh. While it's possible to write a program that doesn't use events, it won't be considered a real Macintosh program. It won't behave like one, and it certainly won't respond like a typical Macintosh program.

Event Types

Events are placed into the event queue in response to three activities: the actions of a user, the operating system, and from other programs. These events are grouped according to event types. Table 1 is a list of common event types that a program can receive and respond to. Not all of them are required or should be used in every program, but some of them, like `DIALOG`, `MENU`, and `MOUSE`, are used in most programs.

TABLE 1. Event types

Name	Event	Description
MENU	user	A menu event is received whenever the user selects a menu item from a program menu.
DIALOG	user/ system	A dialog event can be generated by a user pressing a key, clicking on a window, button, picture, or edit field. System events include window refresh, resume and suspend, cursor position, and many more.
MOUSE	user/ system	A mouse event is received whenever the user clicks the mouse within the contents of a program window. The click must not be in a button or active edit field.
BREAK	user	Received when the user presses the Command-Period keys down.
TIMER	system	Received at a program-specified time interval. Intervals can be defined in terms of seconds or ticks (1/60th of a second).
EDIT	user	Called whenever a program window contains an active edit field and the user is entering characters through the keyboard.
LPRINT	user/ system	Generated during a print operation to halt printing. The user can invoke it by pressing the Command-Period keys, or the print routine can generate it by setting a Print Manager error value.
OVERFLOWS	system	Generated whenever a math operation has exceeded the limits of the variable type.
STOP	system	Generated whenever a program encounters a STOP, END, or QUIT statement in program execution.
EVENT	user/ system	Raw Macintosh events not filtered through *FutureBASIC's* runtime package. Enables you to filter true Mac events before the runtime can examine them.

Now that we know what event types we can expect to receive, how do we go about handling them?

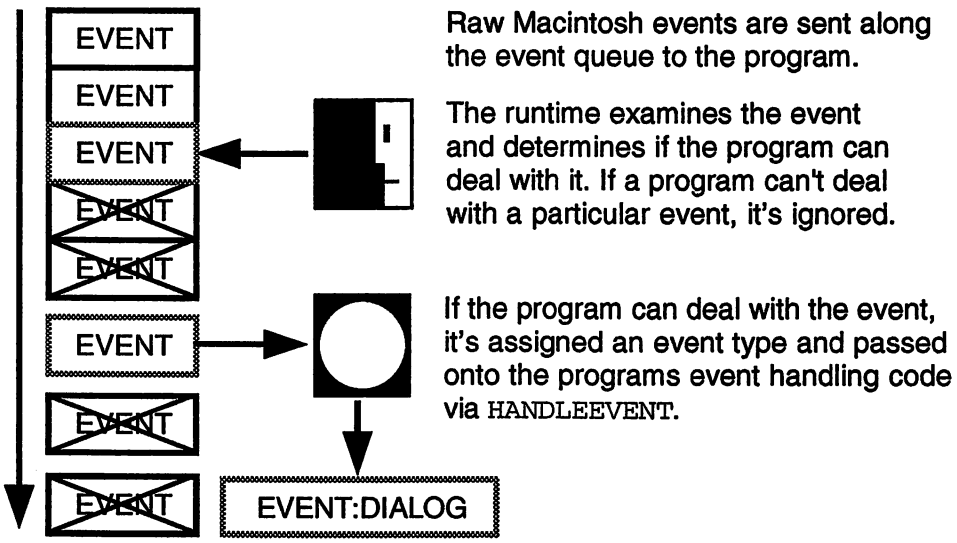
- *In this book, we will only cover events that originate with the user or the operating system.*

Programming for Events

Fortunately, most of the details of handling events is taken care of by the FB runtime. Reread the final event type: EVENT. This is the original raw Macintosh event received by the FB runtime. All other event types should be considered filtered events. This means the runtime has seen the event and massaged it into an easy-to-use form for us.

For example, if the user clicks a button, the raw Mac event _mButDwnEvt is translated by the runtime into a DIALOG event that returns a _btnClick type and the btnID% of the button clicked on. All of the background processing necessary to return that information is handled automatically by the runtime package. Figure 9 graphically shows the process each raw event goes through before the program can handle the event.

FIGURE 9. How events are handled.



Events must be handled as they are received, so it behooves us to have a single point in the program that does nothing but watch for such events. A quick glance back at our program layout shows just such a site, the program's Main Loop section. A single keyword is all that's required to receive events, HANDLEEVENTS.

When HANDLEEVENTS receives an event, it routes it to the appropriate subroutine designed to handle the event. It does this continuously until the program ends. A typical Mac program spends a large percentage of its time simply waiting for events to arrive.

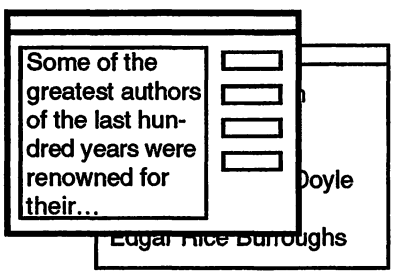
The pseudocode in Program 8 describes the entire event handling sequence in plain English phrases. Its readability helps to clarify the actions required whenever an event is detected.

PROGRAM 8. Event handling pseudocode

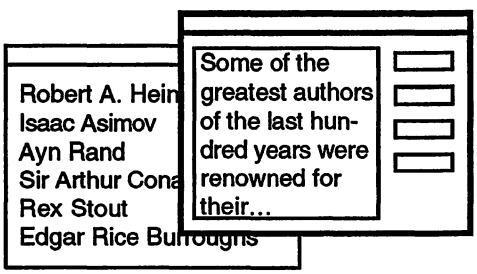
```
Get event from system
Process event in appropriate subroutine
Repeat until program ends
```

Because of a lack of understanding on how events work, many people think they have to grab events as they need them. A typical example is a program that has a separate event loop for each window in the program. This method of event handling is incorrect. By watching for a particular event at a specific time and place, the programmer is ignoring other events that may arrive in the meantime. This restricts the activity of the user, and can cause strange side effects.

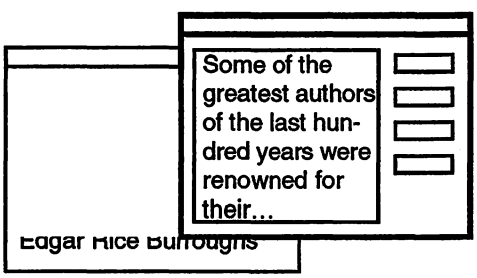
FIGURE 10. Correct and incorrect event handling example.



The user needs to see information in the back window and moves the front window to the right to expose a larger portion of the back window.



A standard event handling program would have little trouble receiving and responding to the refresh event by redrawing the back window's contents.



A program that handles events as it needs them could very well ignore the refresh event and not redraw the window's contents, leaving the user frustrated, unable to continue.

A good example is when a window based event loop watches only for button events in that window. If the user moves the front window, as shown in Figure 10, a refresh event is generated by the operating system, sent to the event queue, received by the runtime, and reported to the program as a `_wndRefresh DIALOG` event. But, since the program was only watching for button events, the refresh event is ignored leaving the uncovered portion of the window blank and the user very annoyed.

Another common problem relates to the same issue. The programmer anticipates a specific event, most notably a `_wndRefresh` event, and so proceeds to redraw a window's contents before the refresh event is received. What happens? The refresh event arrives, the runtime knows you must redraw a portion of the window and kindly erases that portion for you in anticipation of the redraw. In effect, this wipes out what you drew there before the event and leaves it blank. The solution: never anticipate an event. Wait until you receive the `_wndRefresh` event before refreshing a window.

So, the main points for handling events include:

- Create a single main event loop to receive events.
- Respond to a specific event only when it's received (never anticipate the event).
- Respond quickly to the event, then return to the main loop to await the next one.

Now that we know how a program should respond to events, let's look at how we might actually implement some event handling.

Regular Exercise

Let's begin to create our database application that a business might use to maintain an employee list. It won't be entirely suitable in that regard, but it will serve to illustrate one method of creating a full-featured Macintosh application.

The first and most important section of our program is, of course, the Main Loop. Since this portion of code must capture events repeatedly, we'll use a loop block structure. Looking at the program layout, we know to place the loop block in the Main Loop section at the bottom of the program layout. Our Main Loop code is shown in Program 9.

The `DO/UNTIL` loop repeats until the global variable `gQuit%` becomes true. This loop is where the program will spend 99% of its time waiting for the user

PROGRAM 9. Main Loop example.

```
' --- MAIN LOOP ------------------------------------
DO
  HANDLEEVENTS
UNTIL gQuit%
END
```

to choose a menu, click a button, or enter text. The FB statement HANDLEEVENTS is used to remove events from the event queue.

Each event is processed by the program's event handlers. HANDLEEVENTS identifies each event type and directs it to the appropriate event handling routine. Events that a program doesn't handle are ignored by HANDLEEVENTS.[1]

Next, we have a global variable gQuit% (identified by the lowercase "g" as the first letter) we define it in the Globals section of our program like this:

```
' --- GLOBALS --------------------------------------
DIM gQuit%
END GLOBALS
```

Whenever gQuit% is set to true anywhere in the program (remember it's globally available to all subroutines), and when control returns to the Main Loop, the program will exit the DO/UNTIL loop block and end.

Look at the code as it stands in Program 10. It doesn't look like much now, but this small kernel of code is the framework around which we'll construct our entire program.

PROGRAM 10. Event handling loop.

```
' --- HEADER ---------------------------------------
' --- CONSTANTS ------------------------------------
' --- GLOBALS --------------------------------------
DIM gQuit%
END GLOBALS
' --- FUNCTIONS ------------------------------------
' --- MAIN -----------------------------------------
DO
   HANDLEEVENTS
UNTIL gQuit%
END
```

1. An exception being BREAK events. If no Break handler routine is specified, any press of the Command-period key combo will execute a STOP statement, ending the program.

Now that we have some code, it's always smart to test it before moving onto the next section. Remember, the key to top-down programming and stepwise refinement is the testing of code at every level of execution, making sure that it works. Don't wait until later. For our first test, select **Run** from the **Command** menu. FB compiles the program in memory and then executes it. At this point we certainly won't see much, but we can test that we are receiving events. To do this, just press the Command-period keys to stop the program. If everything has gone well, you should be back into the FB editor ready to continue.

Feel free to skip ahead to the Menu chapter. What follows is the first Peak Performance section where some advanced event handling capabilities are explained. Be sure to come back later after you've finished the entire book to explore these features on your own.

Peak Performance

Realistically, *SimpleBase* cannot make use of all the various event types offered by FB. For instance, there is no place in the design of *SimpleBase* to deal with mouse events. Since all of its activity deals with edit fields, buttons, menus, and dialog events, there was nothing left over for the mouse.

To give you some experiences, let's create a small program that will help you understand how mouse events should be handled. Of course, we'll start with the Main Loop, but we'll also add the `ON MOUSE FN` statement to direct mouse events to a mouse handler called `HandleMouse`. The program can be seen in Program 10 (*Mouse Events.main*).

Handling Mouse Events

To remove a mouse event from the event queue, use the `MOUSE(0)` function. This function will return whether the user clicked the mouse button once, twice, or three times. Remember, a mouse event is only reported if the runtime can't assign the click to an active control, edit or picture field, or window title. In most cases, the mouse event will always be either `_click1nDrag`, `_click2nDrag`, or `_click3nDrag`. We can convert it into `_click1`, `_click2`, or `_click3` by using `ABS` on the value returned by `MOUSE(0)`.

- *Due to the nature of FB's mouse handling, you will probably never see a positive `msEvnt%` value. That's because the runtime examines the state of the mouse button when it executes the `MOUSE(0)` function, and due to the greater speed of most machines, you will almost always receive a `_click1nDrag`, `_click2nDrag`, or `_click3nDrag` event.*

PROGRAM 11. Mouse handling program.

```
' --- FUNCTIONS ----------------------------
LOCAL FN HandleMouse
  DIM rect.8, msPt;0, msV%, msH%
  msEvnt% = ABS(MOUSE (0))
  msH% = MOUSE (1)
  msV% = MOUSE (2)
  CALL SETRECT (rect, 100, 100, 200, 200)
  LONG IF FN PTINRECT (msPt, rect)
    SELECT msEvnt%
      CASE _click1                                'show rect
        CALL ERASERECT (rect)
        CALL FRAMERECT (rect)
      CASE _click2                                'invert rect
        CALL INVERTRECT (rect)
      CASE _click3                                'give rect a pattern
        PEN ,,,, RND(31)
        CALL ERASERECT (rect)
        CALL PAINTRECT (rect)
        PEN ,,,, 0
    END SELECT
  XELSE                                           'erase rect
    CALL ERASERECT (rect)
  END IF
END FN
' --- MAIN ---------------------------------
WINDOW 1, "MOUSE Test" : TEXT 3, 9, , 0
ON MOUSE FN HandleMouse
DO
   HANDLEEVENTS
UNTIL 0
END
```

Once the event type is identified, examine the location of the mouse and determine how the program should react. Play with the example and look at how:

- a single click selects a shape,
- a double click inverts the shape,
- a triple click changes the pattern used to fill the shape,
- any click outside of the shape erases it.

The key points to remember when dealing with mouse events:

- A mouse event is only generated when the mouse click doesn't occur in an active window title, edit or picture field, or control.
- The event value of mouse clicks will nearly always be a negative value. Use ABS to convert it into a positive value.
- Clicks are always reported in sequence. In other words, a `_click1` will be received before a `_click2`. If the user double-clicks, the program will receive a `_click1` event, followed by a `_click2` event. It is up to your program to determine what happens on each click.

Cooldown

This chapter has introduced you to events, the key to creating Macintosh programs. In it, we saw how events generated by the user and the operating system are retrieved, processed, and reported to the program, and the importance of waiting for an event before responding. We also learned how the HANDLEEVENTS statement allows us to deal with events by directing each event type to subroutines designed to handle them. In later chapters we'll develop routines to handle specific events. In fact, the next chapter will show you how to implement menus and handle menu events quickly and efficiently.

Menus

Warm-up

This chapter introduces one of the most useful and popular features of the Macintosh: menus. Along the way we'll learn:

- What menus are,
- The different types of menus,
- Various menu features,
- How to create menus, and
- How to respond to menu events.

What are Menus?

Menus are interface elements that allow the user to view or choose from a list of commands. They can appear in any of three different menu styles as shown in Figure 11: pull-down, hierarchical, and pop-up.

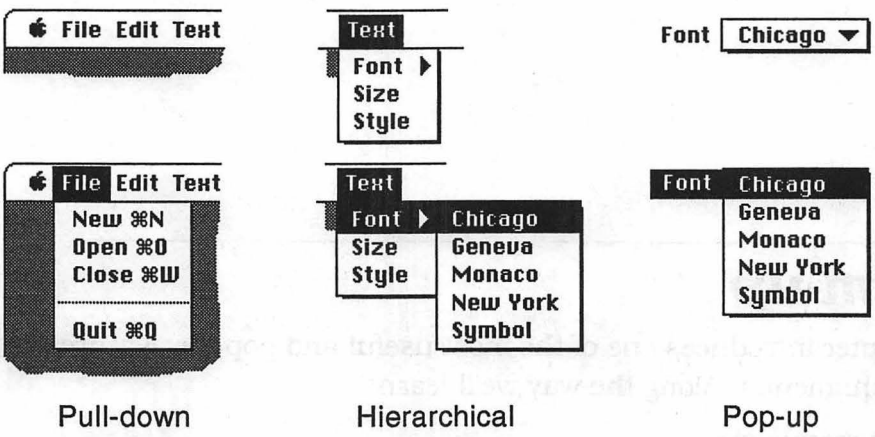
Pull-down menus typically appear at the top of the Macintosh screen. They are usually identified by a menu title, which can be a word or an icon. Pull-down menus appear when their title is clicked on with the mouse. They disappear when the mouse button is released.

Hierarchical menus appear as a sub-menu of a pull-down menu. They are normally identified by a right-pointing arrow that appears at the right edge of

a menu item. Hierarchical menus appear when the user has selected a menu item which contains a sub-menu. In all other aspects, they behave exactly the same as pull-down menus.

Pop-up menus do not appear on the menu bar, but can appear anywhere on the screen when the user clicks the mouse button in a predefined area. Pop-ups are normally identified by a shadowed rectangle containing an item title and a downward pointing arrow. Pop-up menus often have a title that resides to the left of the pop-up itself.

FIGURE 11. The three menu types.



For the *SimpleBase* program we will only deal with pull-down and hierarchical menus. Actually we'll only use pull-downs in our program, but the techniques used for them apply to hierarchical menus, too. See the FB *Handbook* for routines that implement pop-up menus.

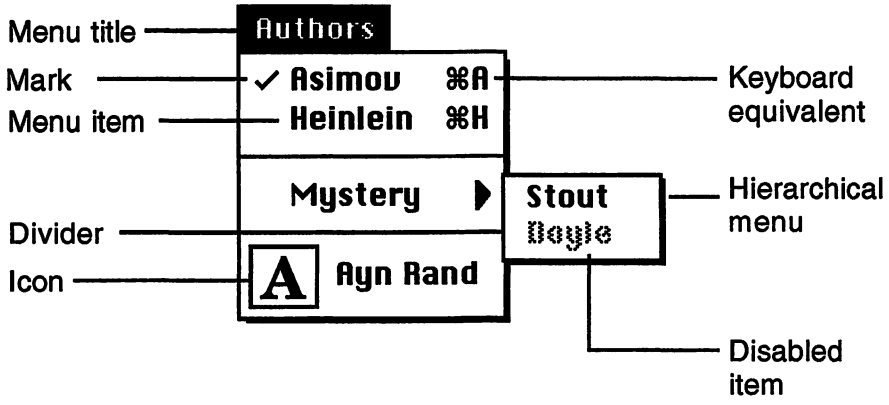
Menu Features

Before we can begin using menus, examine Figure 12 and let's identify some of the common features found on various menus.

Pull-down menus are usually displayed on the menu bar. The **menu bar** is the area at the top of the main screen where all available menu titles are shown. A **menu title** is the text or icon that appears in the menu bar and identifies a particular menu to the user.

When a menu title on the menu bar is clicked, a pull-down menu appears containing a list of choices. Each choice is identified with an item title that describes the action performed by choosing that item. Like a menu title, a **menu item** can be a dividing line, or can consist of text, an icon, or both. A

FIGURE 12. Menu features.



dividing line is an inactive menu item used to separate distinct groupings of menu items.

The user typically uses the mouse to choose a particular item from a menu. Keyboard support is also provided in the form of a keyboard equivalent. A **keyboard equivalent** is a character associated with a particular menu item. By pressing the command (⌘) and character key simultaneously on the keyboard, the user can invoke the menu item.

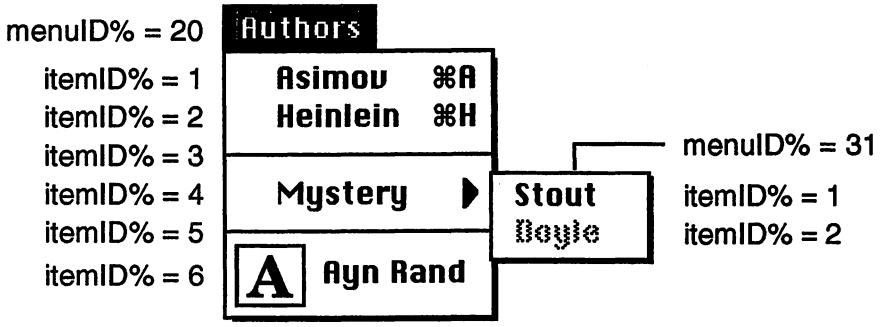
Menus can also contain marks and icons. A **mark** is a character used to indicate a selection in a group of choices, or to indicate the active or inactive states of a particular menu item. Normally, the item mark is a checkmark character, but it can be anything. An **icon** is an image that represents an object, concept, or message.

A menu title or item can also have two states: enabled or disabled. An active menu item can be chosen with the mouse or command key equivalent. A disabled menu item is grayed out and can't be selected by either of these methods. When an entire menu is disabled, its title in the menu bar is grayed out.

A **menu ID** is a unique numerical value assigned by the programmer that identifies the menu to the program. An **item ID** uniquely identifies the menu items for a particular menu. It is also assigned by the programmer to identify the menu item for that menu's item list.

For example, in Figure 13, the **Authors** menu has been assigned a menu ID of 20 by the programmer when the menu was created (we'll see how to do that later). The menu title itself is assigned an item ID of zero, the **Asimov** item has an item ID of one, **Heinlein** is two, and so on. To respond to a selection of

FIGURE 13. Identifying menus and menu items.



the **Ayn Rand** item, the program needs to receive an event that contains a menu ID of 20 and an item ID of 6.

Okay, now that we've described the various features of menus, it's time to learn how to create some of our own.

Creating Menus

There are two ways of creating menus in FB. The first uses the MENU statement to create program menus on the fly. The second requires some setup in *ResEdit* where the menus are stored as MENU resources. This method will be covered in the Peak Performance of this chapter.

The MENU statement requires a minimum of four pieces of information: a menu ID, an item ID, a menu state, and a string for the menu or menu item title. A menu bar on a Mac Classic can comfortably hold 10 menu titles (as long as they aren't extremely long ones), and larger screens can hold more. However, you should always strive to keep menus to a minimum so that all of them will fit on the smallest screen available. Too many menus not only confuses a user into thinking the program is overly complex or difficult to use, but also run the risk of exceeding menu bar space.

To create a new menu, assign it a menu ID, a zero for the item ID, a state setting (in this case _enabled), and the menu's title like this:

```
MENU 1, 0, _enabled, "File"
```

To append an item to the menu, just repeat the MENU statement with a different item ID, state, and item title like this:

```
MENU 1, 1, _enabled, "New"
MENU 1, 2, _enabled, "Open"
MENU 1, 3, _disabled, "-"
MENU 1, 4, _enabled, "Quit"
```

Repeat with different menu IDs for as many menus as required by the program.

The Apple Menu

The menu is a special case when it comes to its menu ID. The menu automatically supports the display and access to user desk accessories and utilities (aliases, applications, etc.) available on the host system.

When the APPLE MENU statement is used, it creates an menu that uses and returns a menu ID of 255. If the menu is created with a MENU resource, it uses and returns a menu ID of 127.

You can append more than one item under an menu using a special separator character (semi-colon) in the menu's title assignment like this:

```
APPLE MENU "About SimpleBase...;Help"
```

You should limit yourself to absolutely no more than three items underneath the menu including: an about window, a help system (if needed), and one more (if required by your program).

The Edit Menu

The second special menu is the Edit menu. The Edit menu provides support for edit fields to cut, copy, paste, and clear text within a program. FB normally assigns it a menu ID of 2. When created as a MENU resource, it should also have a menu ID of 2. For more information on implementing some custom cutting and pasting see the chapter "Edit Menus".

Assigning Command Keys

You assign a command key to a menu item by inserting the "/" character at the beginning or the end of an item title. The first character after the "/" will be inserted as the command key. For example, either of these two forms will assign the "**Q**" key to the **Quit** item of our **File** menu:

```
MENU _mFile, _iQuit, _enable, "/QQuit"
```

or

```
MENU _mFile, _iQuit, _enable, "Quit/Q"
```

Once a command key is assigned to a menu item, it can't be changed using the MENU statement. To do that, you'll need to use the Toolbox procedure SetItemCmd.

Many programs have begun assigning modifier keys like Shift, Option, and Control to menu items. These are not supported by Apple's default menu

TABLE 2. MENU title styles.

Meta-character...	Font Format...
B	Bold
I	Italic
O	Outline
S	Shadow
U	Underline

definition and the means of implementing such keys is beyond the scope of this book.

Assigning Icons

You can assign an icon to a menu item. However, doing so prevents the item from having a command key associated with it. The icon displayed in a menu item must have a resource ID that's within the range of 257 to 512.

You assign an icon to the item using the "^" character followed by the icon's ID number minus 256 in the item title. For example, to display icon #257, use:

```
MENU 1, 1, _enabled, "^1New"
```

To assign icon #258 use:

```
MENU 1, 2, _enabled, "^2Open"
```

Assigning Text Styles

It's possible to provide each menu item with a unique text style using a meta-character. A **meta-character** defines which style the item title will display. This is done by embedding the "<" symbol in the title string followed immediately by the style meta-character. The style settings can be seen in Table 2.

For example, to create a bold faced menu item you would do this:

```
MENU 1, 1, _enabled, "<BNew"
```

However, in most situations, assigning any style other than plain to a menu screams "amateur programmer". If an item not part of a font's **Style** or **Size** menu, don't use a style. There may be a very good reason why a style should be applied to a menu item, but in most cases you can get by without it. The decision is yours.

Unhighlighting Menus

When the user chooses a menu item, the menu title is automatically inverted, or highlighted, to remind the user which menu was chosen. To unhighlight it, use the MENU statement without any parameters. We'll see how to do this once we begin handling menu events later in the chapter.

Enabling & Disabling Menus

One of the most valuable features of menus is that they can guide the user under any situation. They do this by restricting menu and item choices with disabled menus and menu items. For example, if a document isn't open, it would be pointless to allow a user to **Close**, **Save**, or **Print** a non-existent file. By disabling those menu items, the user knows that a file must be open before the choices become available.

This type of user guidance is done using the MENU statement to change the menu state. For example, to disable the **Close** item (item #3) on our **File** menu, we can do this:

```
MENU menuID%, 3, _disable
```

and re-enable it using:

```
MENU menuID%, 3, _enable
```

We can also disable an entire menu using item number zero (the menu title) like this:

```
MENU menuID%, 0, _disable
```

and re-enable it using:

```
MENU menuID%, 0, _enable
```

Marking a Menu Item

You can show that an item is selected by displaying an item mark next to the item title. The mark is usually a checkmark which can be added to the menu by setting the menu item state to _checked like this:

```
MENU menuID%, itemID%, _checked
```

However, the mark can be any character you desire. To mark an item with a non-standard character, set the state of the item to the ASC value of the character to display. For example, to display a bullet (•) character you would do this:

```
MENU menuID%, itemID%, ASC ("•")
```

To add a mark to an item that contains several item titles in a single string, such as an menu, use the "!" symbol before the character like this:

```
MENU menuID%, itemID%, _enable, "New Record;!•Open;!•Close;Quit"
```

To remove a mark, reset the item state to `_enable` using:

```
MENU menuID%, itemID%, _enable
```

Changing Item Titles

It's also possible to change a menu item's title by assigning a new one to the menu item. If the menu is already built, the runtime replaces the current title with the new one. Still using the **File** menu example, we can modify the **New** item by executing a line like this:

```
MENU menuID%, itemID%, _enable, "New Record"
```

And change it back to its original form using:

```
MENU menuID%, itemID%, _enable, "New"
```

We'll see how we can use this technique later, not only to create new employee database files, but also to create new records once a file is open.

Deleting Menus

You normally don't have to worry about removing menus built in your program. The standard FB runtime takes care of that task for you whenever the program ends. This is another benefit of using the FB runtime package.

Note that resource menus having IDs in the range of 1 to 31 are also automatically deleted at the program end by the FB runtime.

Regular Exercise

Now that we understand what menus are, let's begin adding them to our program and learn how to respond to their selection.

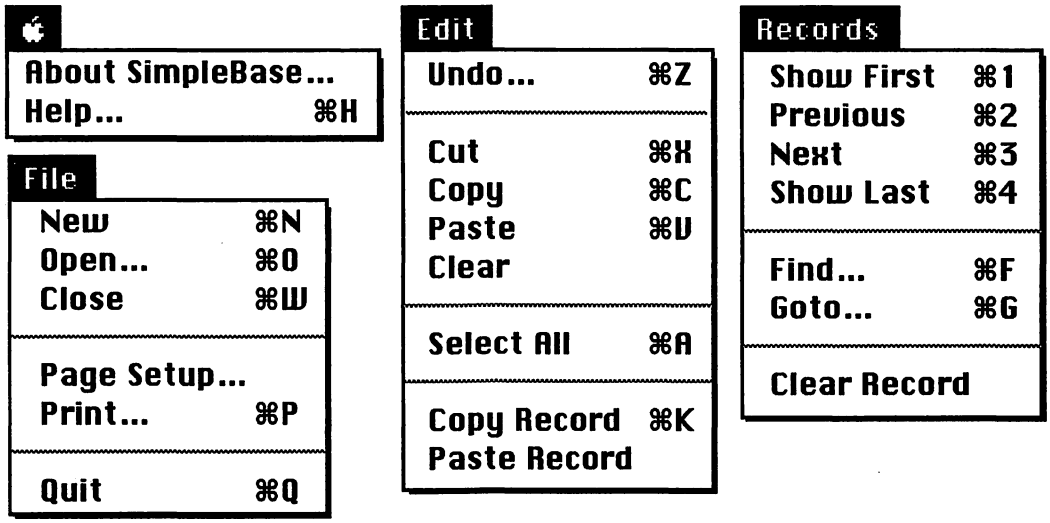
Program Menus

Creating menus is a two step process. First, determine what menus the program will need, then define them in syntax FB will understand. What could be easier? Our goal is to create menus that appear just like those shown in Figure 14.

Step One: Menu Constants

The first step to creating our menus is determining which ones the program will need. Since *SimpleBase* will deal with files, we need a **File** menu, and of course, it uses text, so an **Edit** menu is required. Additionally, we'll want to be able to move around and manipulate the various records in our database, so

FIGURE 14. SimpleBase menus.



a **Records** menu would be nice. Finally, we should always have an menu so users will have access to their desk accessories and system utilities.

The easiest place to define our menus is in the Constants section of the program. By defining an equate for each menu and menu item, we begin the process of creating self-documenting code. For example, it's much easier to read and understand a program line like this:

```
MENU _mfile, _iQuit, _enable
```

than something like this:

```
MENU 1, 10, 1
```

Who can remember what item #10 is, especially if six months have gone by since you last worked on the source code? Make it easy on yourself, using techniques like these will make your source code almost self-documenting.

Looking at the menus again, we see that we need to create constants for our four program menus: , **File**, **Edit**, and **Records**. The menu has already been predefined with a constant (_appleMenu), so we only have to create the others. For mnemonic ease, each menu constant is preceded by a lower case "m", while each item constant uses a lower case "i" as an identifier.

We define the menu constants like this:

```
_mFile = 1
_mEdit = 2
_mRecord = 3
```

PROGRAM 12. Defining menu equates.

```
'  --- EQUATES ---------------------------------------
' >>> APPLE MENU
_iAbout      = 1
_iHelp       = 2
' >>> FILE MENU
_mFile       = 1
_iNew        = 1
_iOpen       = 2
_iClose      = 3
' -------------
_iPageSetup = 5
_iPrint      = 6
' -------------
_iQuit       = 8
' >>> EDIT MENU
_mEdit       = 2
_iUndo       = 1
' -------------
_iCut        = 3
_iCopy       = 4
_iPaste      = 5
_iClear      = 6
' -------------
_iSelectAll = 8
' -------------
_iCopyRec = 10
_iPasteRec = 11
' >>> RECORD MENU
_mRecord     = 3
_iFirstRec   = 1
_iPrevRec    = 2
_iNextRec    = 3
_iLastRec    = 4
' -------------
_iFindRec    = 6
_iGotoRec    = 7
' -------------
_iClearRec   = 9
```

With our main menus decided upon, we can define their respective menu items in the same way. The final list is shown in Program 12. Now that we have the menu constants defined, it's time to add them to our program.

Step Two: Menus

When adding an entire program of menus, it's always tempting to bundle the entire creation process into the `Initialize` subroutine of the program. Don't do it. In the spirit of top-down design, you should always break each routine down to its smallest parts, hiding the details until the very last routine. With that idea, we'll add our menus to the `Initialize` routine with one line of code as shown in Program 13.

PROGRAM 13. Adding menus to Initialize.

```
LOCAL FN Initialize
   FN BuildMenus
END FN
```

You can see what this routine is doing for the `Initialize` routine, it's building our program menus. Now, let's add the function `BuildMenus` where all of the dirty work is done. The entire `BuildMenus` routine can be seen in Program 14.

We start by defining the menu items using the `APPLEMENU` statement. `APPLEMENU` accepts a string containing one or more item titles. Each item title is separated by a semi-colon. In our case we'll use two, one for the about information item, and one for program help.

Next, the **File** menu provides all the commands required to handle *SimpleBase's* data files. We assign the standard **New**, **Open**, **Close**, **Page Setup**, **Print**, and **Quit** menu items normally associated with the **File** menu. Each allows us to perform actions on the database as a complete unit. You may notice that there is no **Save** or **Save As...** items. We handle saving automatically, so users never have to worry about losing their data. We'll see how we implement these features when we create some files.

The **Edit** menu is another special case menu handled by the FB runtime. Here we create an entire **Edit** menu containing all the common editing items like **Undo**, **Cut**, **Copy**, **Paste**, etc., then add three additional items for: **Select All**, **Copy Record**, and **Paste Record**.

The final menu, **Records**, contains items related to moving around our database file once it's open. We have six items: **Show First**, **Previous**, **Next**,

Show Last, **Find...**, and **Goto...**. In addition, we enable the user to erase data with **Clear Record.**

You may have noticed the ellipsis (...) that appears on some menu items but not others. By design, an ellipsis tells the user that selecting this item requires additional information in order to complete the requested task. Usually, a window appears asking for that information. For example, the **Find** item uses the ellipsis in the title because it will ask for some search text from the user.

Note that we can safely skip all blank menu items, or dividing lines, in our menu definitions, since FB automatically fills them in for us.

That's all there is to building menus. If you run the program at this stage, you'll see the menus, but they don't respond like a menu should. We'll take care of that now.

PROGRAM 14. BuildMenus routine.

```
LOCAL FN BuildMenus
  APPLE MENU "About SimpleBase…;Help…/H"
  MENU _mFile, 0          , _enable , "File"
  MENU _mFile, _iNew      , _enable , "New/N"
  MENU _mFile, _iOpen     , _enable , "Open…/O"
  MENU _mFile, _iClose    , _disable, "Close/W"
  MENU _mFile, _iPageSetup, _disable, "Page Setup…"
  MENU _mFile, _iPrint    , _disable, "Print…/P"
  MENU _mFile, _iQuit     , _enable , "Quit/Q"
  EDIT MENU _mEdit
  MENU _mEdit, _iSelectAll , _enable, "Select All/A"
  MENU _mEdit, _iCopyRec   , _enable, "Copy Record/K"
  MENU _mEdit, _iPasteRec  , _enable, " Paste Record"
  MENU _mRecord, 0          , _disable, "Records"
  MENU _mRecord, _iFirstRec , _enable , "Show First/1"
  MENU _mRecord, _iPrevRec  , _enable , "Previous/2"
  MENU _mRecord, _iNextRec  , _enable , "Next/3"
  MENU _mRecord, _iLastRec  , _enable , "Show Last/4"
  MENU _mRecord, _iFindRec  , _enable , "Find…/F"
  MENU _mRecord, _iGotoRec  , _enable , "Goto…/G"
  MENU _mRecord, _iClearRec , _enable , "Clear Record"
END FN
```

PROGRAM 15. Menu event handling pseudocode.

```
Get a menu event
Determine the correct menu and item ID numbers
Call the routine to handle the selected menu
The routine deals with the event
Return to look for the next event
```

Handling Menu Events

I'll bet you thought we would never get back to talking about events. They haven't been forgotten, they were just set aside while we got our menus in place and ready to go. So now let's talk about menu events.

As mentioned in the chapter "Events", every time a user chooses a menu item, an event is generated. The Main Loop sees the raw event, translates it into a menu event, then passes it onto the program's menu event handling routine. The pseudocode to handle menu events is shown in Program 15:

The activities described in this particular pseudocode example are spread over several different subroutines in the program. Each decision that deals with the event passes control to the next stage of the design. The Main Loop gets the event. It calls the assigned menu handling routine which extracts the menu ID and item ID from the event. The menu handling routine then calls the subroutine designed to deal with the menu selection. The subroutine may in turn call other subroutines to deal with the event. When finished, control returns to the Main Loop to await the next event.

Remember that `HANDLEEVENTS` has the job of retrieving the event from the event queue and handing it over to the program's appropriate event handler. But how does `HANDLEEVENTS` know which handler that is? Simple: we tell it like this:

```
ON MENU FN MenuEventHandler
```

We add this line of code just prior to entering the Main Loop of the program. All `ON <event> FN` statements are, in effect, a sign that points `HANDLEEVENTS` to the function designed to deal with the event. In this case, it tells `HANDLEEVENTS` to direct all menu events to the routine `MenuEventHandler`. Once again, we bury the details of handling menu events in this subroutine. The menu event handling subroutine is shown in Program 16.

There are two pieces of information the program requires from the event: the menu and the item number, often referred to as the menu ID and item ID. These two pieces of data are used to identify the menu selection made by the user. The `MENU` function returns both values, placing them in appropriately

PROGRAM 16. Skeleton menu handling routine.

```
LOCAL FN MenuEventHandler
  menuID% = MENU (_menuID)
  itemID% = MENU (_itemID)
  MENU
END FN
```

named variables called menuID% and itemID%. The final MENU statement (without parameters) in the routine, unhighlights the chosen menu title. This should occur after the menu handling subroutine has completed executing.

Okay, our program has received a menu event, and extracted the correct menu and item values with the MENU function — what's the next move? Well, at this point of top-down design we hide the details in another subroutine, or in this case, routines. In other words, we create a subroutine for each menu in the program and call it when the correct menuID% appears. The best way to call multiple items from a single value is to use the SELECT/END SELECT structure. Program 17 shows what our complete menu handling function looks like.

PROGRAM 17. Enhanced menu handling routine.

```
LOCAL FN MenuEventHandler
  menuID% = MENU (_menuID)
  itemID% = MENU (_itemID)
  SELECT menuID%
    CASE _appleMenu : FN DoAppleMenu (itemID%)
    CASE _mFile : FN DoFileMenu (itemID%)
    CASE _mEdit : FN DoEditMenu (itemID%)
    CASE _mRecord : FN DoRecordMenu (itemID%)
  END SELECT
  MENU
END FN
```

Notice how the predefined menu constants make it easy to understand which menu is called. Also note the descriptive routine names that leave little doubt as to their defined task. Each subroutine handles its own menu items. We make sure to pass the itemID% to them since each will need that information to make their own internal decisions. Let's examine a couple of them to see how they do that.

The first is DoAppleMenu which is shown in Program 18. Since it has to make a choice of which item was selected, we pass it the itemID% as a parameter.

We use another SELECT/END SELECT structure to call the final routines, the ones which will actually execute the chosen menu task.

PROGRAM 18. Skeleton apple menu routine.

```
LOCAL FN DoAppleMenu (itemID%)
  SELECT itemID%
    CASE _iAbout : FN ItemAbout
    CASE _iHelp : FN ItemHelp
  END SELECT
END FN
```

Since the menu can also contain special items like desk accessories, or with System 7, aliases, applications, and documents, you might think we have some special processing to do. Wrong! We don't have to worry about that at all since the FB runtime takes care of those details for us. We deal with the menu items we specified with APPLE MENU. No muss, no fuss.

The next menu handler to examine is the **File** menu as shown in Program 19. Again, we use SELECT/END SELECT to choose between all the possible values of itemID% and call to the appropriate item handling routine. This same technique is used for the **Edit** and **Record** menus.

PROGRAM 19. File menu routine.

```
LOCAL FN DoFileMenu (itemID%)
  SELECT itemID%
    CASE _iNew : FN ItemNew
    CASE _iOpen : FN ItemOpen
    CASE _iClose : FN ItemClose
    CASE _iPageSetup : FN ItemPageSetup
    CASE _iPrint : FN ItemPrint
    CASE _iQuit : FN ItemQuit
  END SELECT
  MENU
END FN
```

PROGRAM 20. Skeleton Print item routines

```
LOCAL FN ItemAbout
  PRINT "About item"
END FN

LOCAL FN ItemHelp
  PRINT "Help item"
END FN
```

Handling Menu Selections

At this time, it's possible to add all of the individual item handlers for each menu. Sometimes called **skeleton** routines, these functions are fully callable but perform little or no actual work. Functionality will be added later as we develop *SimpleBase*.

We can check to see that our menus work by adding a single `BEEP` or `PRINT` statement into each one and testing it. For example, we can test the menu subroutines by adding `PRINT` statements as shown in Program 20.

If you run the program at this time, you can select either **About SimpleBase...** or **Help...** from the menu. If the program encounters no errors, the correct skeleton message will appear in the program window when an item is chosen from the menu.

Not all of the routines have to be skeletons at this point. One in particular, **Quit**, is quickly implemented. The Main Loop is constantly checking the value of `gQuit%`, so set `gQuit%` to `_true` (i.e. to anything but zero) in order to exit our program. We do that in the `ItemQuit` routine like this:

```
LOCAL FN ItemQuit
  gQuit% = _true
END FN
```

Here, shown in all its beginning glory in Program 21, is the remainder of the program (consisting mostly of skeleton routines) to handle all *SimpleBase* menu items.

PROGRAM 21. Menu item handlers.

```
' ____________________________________ APPLE MENU ITEM HANDLERS

LOCAL FN ItemAbout
END FN

LOCAL FN ItemHelp
END FN

LOCAL FN DoAppleMenu (itemID%)
  SELECT itemID%
    CASE_iAbout : FN ItemAbout
    CASE _iHelp : FN ItemHelp
  END SELECT
END FN

' ____________________________________ FILE MENU ITEM HANDLERS

LOCAL FN ItemNew
END FN

LOCAL FN ItemOpen
END FN

LOCAL FN ItemClose
END FN

LOCAL FN ItemPageSetup
END FN

LOCAL FN ItemPrint
END FN

LOCAL FN ItemQuit
  gQuit% = _true
END FN

LOCAL FN DoFileMenu (itemID%)
  SELECT itemID%
    CASE _iNew : FN ItemNew
    CASE _iOpen : FN ItemOpen
    CASE _iClose : FN ItemClose
    CASE _iPageSetup : FN ItemPageSetup
    CASE _iPrint : FN ItemPrint
    CASE _iQuit : FN ItemQuit
  END SELECT
END FN
```

continued on next page...

```
' _____ EDIT MENU ITEM HANDLERS

LOCAL FN ItemUndo
END FN

LOCAL FN ItemCut
END FN

LOCAL FN ItemCopy
END FN

LOCAL FN ItemPaste
END FN

LOCAL FN ItemClear
END FN

LOCAL FN ItemSelectAll
END FN

LOCAL FN ItemCopyRecord
END FN

LOCAL FN ItemPasteRecord
END FN

LOCAL FN DoEditMenu (itemID%)
  SELECT itemID%
    CASE _iUndo : FN ItemUndo
    CASE _iCut : FN ItemCut
    CASE _iCopy : FN ItemCopy
    CASE _iPaste : FN ItemPaste
    CASE _iSelectAll : FN ItemClear
    CASE _iCopyRec : FN ItemCopyRecord
    CASE _iPasteRec : FN ItemPasteRecord
  END SELECT
END FN
' _____ RECORD MENU ITEM HANDLERS

LOCAL FN ItemShowFirst
END FN

LOCAL FN ItemPrevRecord
END FN

LOCAL FN ItemNextRecord
END FN

LOCAL FN ItemLastRecord
END FN
```

continued on next page...

```
LOCAL FN ItemLastRecord
END FN

LOCAL FN ItemFindRecord
END FN

LOCAL FN ItemGotoRecord
END FN

LOCAL FN ItemClearRecord
END FN

LOCAL FN DoRecordMenu (itemID%)
  SELECT itemID%
    CASE _iFirstRec : FN ItemFirstRecord
    CASE _iPrevRec : FN ItemPrevRecord
    CASE _iNextRec : FN ItemNextRecord
    CASE _iLastRec : FN ItemLastRecord
    CASE _iFindRec : FN ItemFindRecord
    CASE _iGotoRec : FN ItemGotoRecord
    CASE _iClearRec : FN ItemClearRecord
  END SELECT
END FN
```

Peak Performance

If you haven't done so already, please read the chapter "Resources" on using *ResEdit*. If you are already familiar with *ResEdit*, welcome, you're about to learn how to add resource menus to your programs.

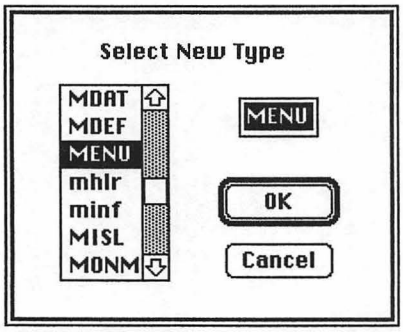
Resource Menus

Adding menus as resources is almost easier than using the MENU statement. Because , **File**, and **Edit** menu tend to appear in every Mac program, you can create them once and just copy them into each new project. Additionally, they are much easier to change both during program development and after compilation. Should you decide to change an item name, just drop into *ResEdit* and make the change, no need to recompile.

Creating MENU Resources

We start by opening the file *SimpleBase.rsrc*. Next, choose **Create New Resource** from the **Resource** menu and select the MENU type in the scrolling list as shown in Figure 15. Click **OK**. *ResEdit* automatically creates a new menu for you and opens the Menu editor shown in Figure 16. Here is where you enter the menu items, add command keys, colors, and styles.

FIGURE 15. Select New Type dialog.



On the left of the editor you can see a small display of the current menu. As items are added they will be appended to the list. On the right, enter the title for the menu item as well as setting if it should be initially enabled. On the bottom-right are selections for adding color to the various features of the menu. On the menu bar itself is a real-time copy of your menu that you can check out as you add items so you know exactly how it will appear.

Additionally, you can add hierarchical menus to menu items by choosing the **has Submenu** checkbox and inserting a menu ID number. Since we don't need any hierarchical menus, we'll just skip that feature for now.

While in the Menu Editor, choose **Get Resource Info** from the **Resources** menu. The MENU information window shown in Figure 17 appears. Here you can enter the menu ID used by your program to identify the menu. In the case of the **File** menu, that ID is 1. I also like to enter the menu name for easier identification later.

FIGURE 16. MENU resource editor.

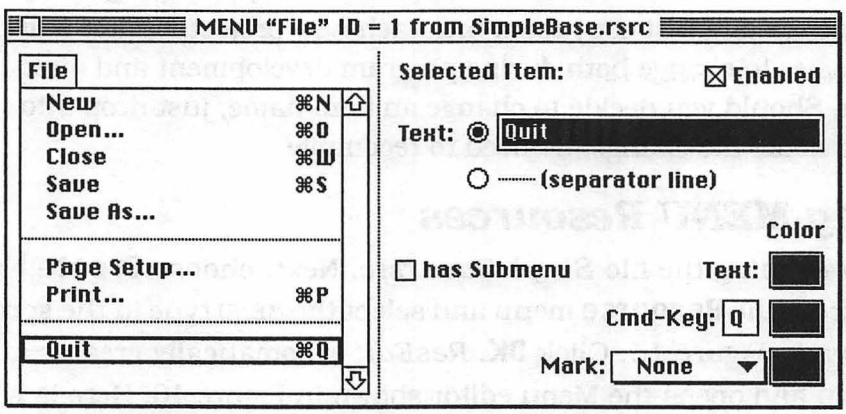
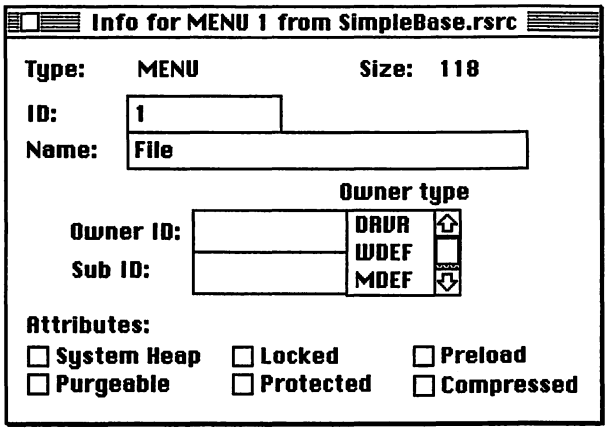


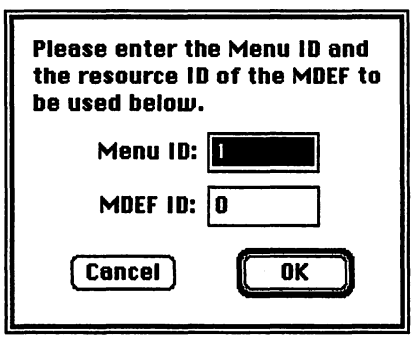
FIGURE 17. MENU resource information.



- *Warning, unlike other resources, never make a* MENU *resource purgeable. If you do, the menu may get purged by the Memory manager when it needs space, and the next time the user selects the menu a system error (-84) will result.*

When you close the resource info window, *ResEdit* warns you of the menu ID change with an alert. Just ignore its advice and click **OK**. Failure to change a menu's ID isn't serious, you just won't be able to use it in your program. It's the menu ID number that the FB runtime uses to identify a menu selection. If the **File**'s MENU resource has a menu ID of 128 and you expect to see 1 in your event loop, it won't happen. However, should you forget to change a menu ID to match the resource ID, just go back into *ResEdit*, open the correct MENU resource, then choose **Get Menu ID** from the **MENU** menu to get the dialog shown in Figure 18.

FIGURE 18. MENU ID dialog.



Do not change the MDEF ID in this dialog unless you know what you're doing. This is the procedure code that tells the system which MDEF resource (Menu DEFinition) to use when drawing the menu. MDEF 0 is the default pull-down menu procedure. Unless you are using a custom MDEF to draw your menu, never change this value.

Repeat the above steps for each menu resource required by the program. For *SimpleBase* that is MENU resources for , **File**, **Edit**, and **Records**. Make sure that the menu ID for each menu matches the resource ID (and the menu should be 127). Forgetting this will prevent FB from finding the correct menu when a user selects it.

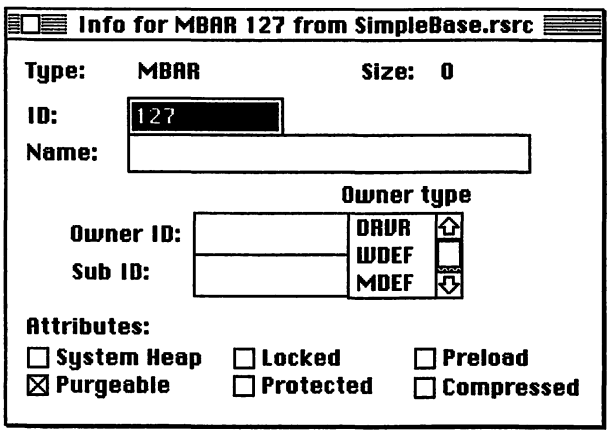
Create MBAR Resource

The next thing to do is add an MBAR resource with an ID of 127 to the resource file. When FB sees this resource type in a program's resource file, it automatically loads it into memory, and then loads the MENU resources listed in it. Voilá, instant program menus with no code.

To create an MBAR resource, start by choosing **Create New Resource** from the **Resource** menu. Enter or scroll to select MBAR from the list of file types just as we did with the MENU resource. Click **OK**.

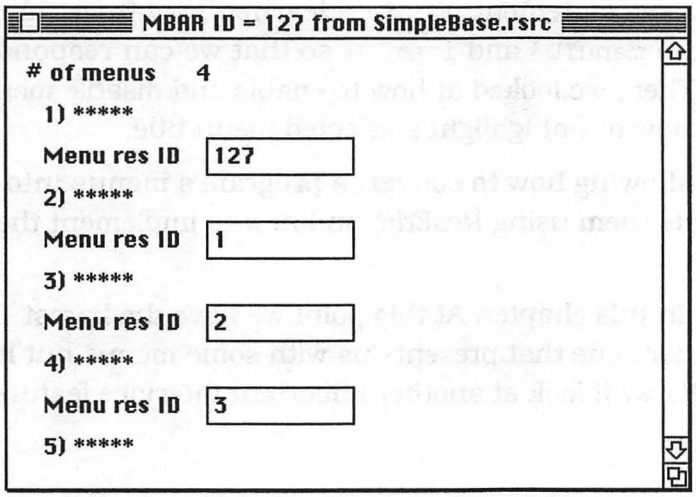
Second, highlight the MBAR resource and choose **Get Resource Info** from the Resource menu. Enter the value 127 in the ID field and check the **Purgeable** box as shown in Figure 19. Close the window by clicking the window's close box.

FIGURE 19. Creating a MBAR resource.



Next, enter the values of the MENU resources the MBAR resource will display on the menu bar. Click on the asterisks shown in the MBAR editor in Figure 20. A box will appear around them. Now choose **Insert New Field** to add a place for a menu ID. Repeat until you have four spaces. Finally, enter your menu ID numbers in the order they should appear on the menu bar.

FIGURE 20. MBAR resource editor.



Finally, save all your work and close the resources file. If it's not already present in your main file, add a RESOURCES statement to the Header section of the file. Go to the `Initialize` subroutine and delete the call to `BuildMenus`. Run the program. If no errors are present in the resource file, your menus will appear and work just as before, with the exception that you didn't use a single line of code to add them to your program.

Cooldown

This chapter covered everything you need to know about menus including: the three types of menus (pull-down, hierarchical, pop-up), and their distinctive features. We looked at one method of creating menus for a program that required us to first define the menu constants, then construct the menus using the MENU statement. We then learned how to get menu events, and recover the `menuID%` and `itemID%` so that we can respond to specific user choices. Then, we looked at how to enable and disable menus and menu items, and how to unhighlight a selected menu title.

Finally, we closed by showing how to convert a program's menus into MENU resources, how to create them using *ResEdit*, and how to implement them using a MBAR resource.

We learned quite a bit in this chapter. At this point we have the barest skeleton of an application, one that presents us with some menus but little else. In the next chapter we'll look at another important interface feature, the window.

CHAPTER 5

Windows

Warm-up

This chapter introduces you to another common interface component, windows. In this chapter you will:

- Learn what windows are,
- Identify common window types,
- Identify window features, and
- Learn how to create windows.

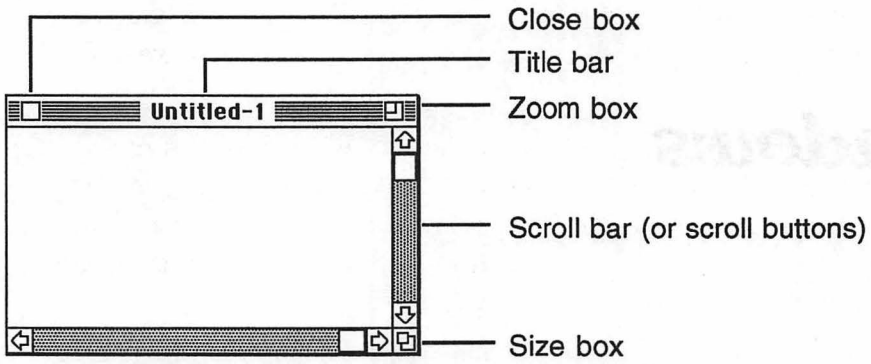
What are Windows?

A **window** is a specified area of the screen that allows the user to view or enter information. A program can have multiple windows on the screen, each performing a different function.

Programs typically create windows that allow the user to enter data, or view text, graphics, or other information in a meaningful way. When displaying a document, the window provides a view into the document contents. The user can change, move, resize, and close windows. There are a number of standardized window elements that make using windows convenient for the user. These standard elements are shown in Figure 21.

The **close box** enables the user to close a window. This releases all data structures related to the window or its contents including buttons and edit fields.

FIGURE 21. Window features at a glance.

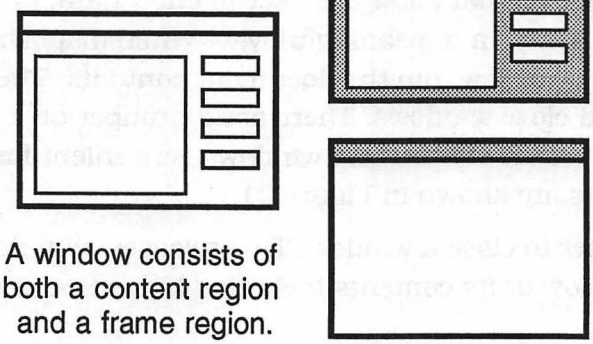


The **title bar** often displays the name of the document and indicates whether the window is active or inactive. It also allows the user to reposition the window by dragging the title bar with the mouse. Some windows do not have title bars (see the chapter "Alerts") and cannot be moved by the user.

The **zoom box** allows the user to switch the window's size between two predefined sizes, while the **size box** enables the user to resize the window dimensions manually.

Scroll bars (scroll buttons in FB terminology) enable the user to see other portions of the document when the data in the document exceeds the viewable area of the window. Scroll bars are not part of the window, but are a control placed there by the program (see the chapter "Scroll Buttons" for more information).

FIGURE 22. Window content and frame regions.



A window consists of both a content region and a frame region.

The content region of a window contains buttons, edit and picture fields, as well as grow boxes.

The frame region of a window consists of the title bar, close box, zoom box, and frame around the window.

A window as shown in Figure 22 consists of two main parts: the content region and the frame. The **content region** is that area of the window in which a program can display data, controls, and the size box. The **frame** is the rest of the window including the title bar, the close and zoom boxes, and the window's outline.

For the most part, the FB runtime handles all of the actions required to manage windows in your programs. The programmer, must define the window's initial size, and close it, but moving, resizing, and zooming are all handled automatically. Also, the runtime allows a program to handle any of these actions if necessary.

Creating Windows

Windows are the main method a program uses to display output and receive input from a user. Since a program can require many different windows, FB provides support for up to 63 program windows. Each is uniquely identified by a `wndID%` number that ranges from 1 to 63. A specific window is built using the `WINDOW` statement.

- *Never use a `wndID%` of zero as that is reserved for the default Command window.*

The `WINDOW` statement needs several pieces of information, a `wndID%`, a title (if required), a rectangle that specifies its position on the screen, a window type, and while not mandatory, a window class.

```
WINDOW #wndID%, "MY WND", (10,50) - (200,80), _docNoGrow, wndClass%
```

The `wndID%`, of course, specifies which window to build. If `wndID%` is proceeded by a negative sign, the window is built invisibly on the screen. This enables us to build our data entry window with all of its fields and buttons beyond the user's view, then display it when complete. Using this technique tricks the user into thinking that the program builds windows quickly. This is because since the window appears to leap onto the screen all at once, instead of a piece at a time.

- *Don't be confused by the negative sign on `wndID%`, the `wndID%` is always a positive value. The negative sign is a flag to the runtime that tells it to build the window invisibly.*

The title in windows with title bars can be up to 255 characters, but should be smaller than that if you expect all of it to be displayed.

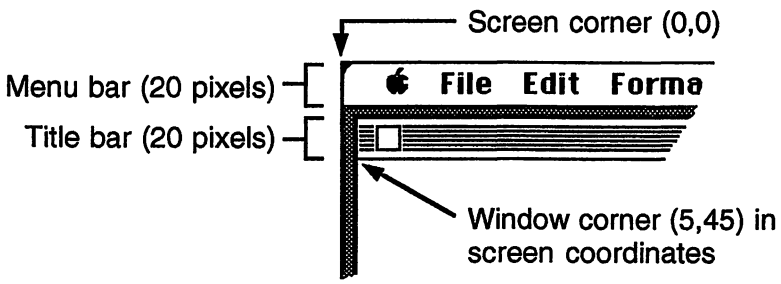
The window rectangle specifies the size of the window as well as its location on the screen. It has the following format:

```
(left, top) - (right, bottom)
```

for its two opposite corners, where the left-top corner of the screen is 0, 0 and the right-bottom corner can range from 512, 342 and beyond depending on the size of the monitor. This is important to remember, a default window should never be larger than the Classic's 9" monitor. This means that controls and fields should always appear so that they are available even on the Macintosh, the smallest screen. Forcing the user to scroll fields or controls into view is unacceptable in a program. Windows should take advantage of larger screens if available, but should never force users of smaller screens to perform mouse or command key gymnastics to accomplish normal tasks in a window.

The top-left coordinates of a window start at the top-left corner of its content region, not the title bar. Therefore, when positioning a window that has a title bar, always add 20 pixels to the window top to compensate for the title bar. In addition, the menu bar adds another 20 pixels to the top position or else the window's title bar will be obscured.

FIGURE 23. Window positioning on the screen.



Centering Windows

You center a window on a screen by specifying the left and top coordinates of the window rectangle to 0, 0.

When Macs only had small 9" screens, only alerts and dialogs got centered on the screen. When larger monitors came out, it became important to center windows instead of restricting them to the upper left corner of the screen. In the beginning, programmers had to calculate this center for each window, today the FB runtime has this capability built into it. To center any window on any screen, just do this:

```
WINDOW #wndID%, "CENTERED WND", (0, 0) - (200, 80), _docNoGrow
```

One caveat, however, never rebuild a centered window without first closing it. If you fail to close the window, the runtime will recalculate the center and the window will appear in the upper-left corner of the screen with only a quarter of it visible.

You can determine the screen size using the SYSTEM functions:

```
scrnWidth% = SYSTEM (_scrnWidth)
scrnHeight% = SYSTEM (_scrnHeight)
```

The maximum screen size in both directions extends from -32768 to +32767, or about 35 feet in height and width. This is plenty of room for maneuvering windows.

Window Types

Each window has a purpose. Some are used for normal documents, others provide information or alert the user to a problem. Still others are used to request input from the user. The window types you use will depend on the program you are writing. For *SimpleBase* we rely on the window type _docNoGrow for most of our windows.

Some of our windows could have probably used the standard _dialog type of window, especially the Find and Goto windows. The decision not to use them is because of System 7. Since multiple applications can be open, multiple windows can appear on the screen and create window confusion. I felt it more important to display window titles to identify the window, something the _docNoGrow type supports but _dialog does not. You may choose differently.

Table 3 contains a list of various window types available and a brief description of their common uses in Macintosh programs.

Modal vs. Non-Modal Windows

There are two main kinds of windows, non-modal and modal. Non-modal are more popular and enable users to work within the window, choose menus, click in other windows, and generally work in any order they wish. For example, the following WINDOW statement builds a regular non-modal window:

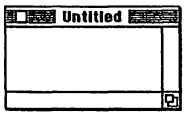
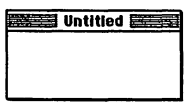
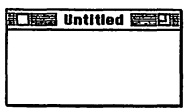
```
WINDOW #wndID%, "NON-MODAL", (0, 0) - (500, 300), _docNoGrow
```

A modal window, however, is one that prevents the user from clicking or selecting anything outside of the window. It generally appears as a dialog or alert window type. Any attempt to click outside of the window results in a beep. This type of window is created when the window type in the WINDOW statement is proceeded by a negative sign. The following WINDOW statement builds a modal window:

```
WINDOW #wndID%, "MODAL", (0, 0) - (500, 300), _docNoGrow
```

In the age of System 7, modal window types are generally frowned upon in programs. Many different applications can be open at once, and a modal window prevents the user from switching out of the current application. It's

TABLE 3. Window Types

WINDOW	DESCRIPTION
	*Document* — the standard Macintosh window containing a title bar, vertical and horizontal scroll bars, as well as a size box. Used by most applications for their documents.
	*Dialog* — a non-moveable window that normally appears requesting additional information or displaying an alert message.
	*Plain* — another form of a non-moveable dialog window. This one gets occasional use as a start-up window in programs.
	*Shadow* —yet another form of the non-movable dialog window. This one is also used occasionally as a start-up window in programs.
	*No Grow Document* — a special form of the document window, this version doesn't contain scroll bars or a size box. Occasionally used for non-modal dialog windows before the Moveable Dialog type appeared.
	*Moveable Dialog* — introduced with System 7, this form of dialog window enables the user to position the window anywhere on the screen rather than an arbitrary position defined by the programmer.
	*Zoom Document* — a standard document window with a zoom box. If the document window can be resized, it should include a zoom box to enable the user to quickly zoom the window to full size or down to its minimum size without clicking and dragging the size box.
	*NoGrow Zoom Document* — a variation of the normal no grow document window. This version has a zoom box to toggle the window between two defined sizes.
	*Round Document* — this seldom used window type was originally prescribed for desk accessories, but never really caught on.

strongly recommended that you never use modal windows created by WINDOW in your program. Instead, see the chapter "Alerts" for ways of using alert windows instead.

Window Classes

The final parameter for a WINDOW statement is the window class. This is a programmer specific value that has one important use. It allows a number of windows to belong to a class. This class identifier is important because it enables the programmer to write routines that deal with a specific window class instead of a specific window number.

What this means is that it is possible to write routines to support a class of windows and have them work for all windows of that class. Once *SimpleBase* is up and running for one employee file, we can create additional data entry windows, and have the same subroutines work for all of them.

Later we'll see how to use the window class to determine all of the actions required for a window, from building to refreshing to closing.

Hiding & Showing Windows

Once a window is built, it's possible to hide it or show it to the user. There are many reasons for wanting to hide a window from the user. It may be that you have an extremely complex window that takes a while to build each time. By hiding it instead of closing it, it can appear much faster the next time the user calls for it. All of the windows in *SimpleBase* are built invisibly, then pop onto the screen when complete.

To show a window that was built invisibly to the user, use the WINDOW statement like this:

```
WINDOW #wndID%
```

To make the window visible and directs all text and graphic commands to it. Then, to hide a visible window, just use a negative sign before the wndID% value like this:

```
WINDOW #-wndID%
```

Window Sizing

There are two statements that enable you to control the minimum and maximum size of a re-sizeable window. Users can manually resize the _doc and _docZoom window types, but they shouldn't be able to make them too small, nor too large. Your program can control these sizes using the MINWINDOW and MAXWINDOW statements. For example, to keep a window from becoming too small just do this:

```
MINWINDOW 200,100
```

This sets the minimum size that the runtime will allow all program windows to be reduced to. This is a global setting and effects all resizeable windows in the program. On the flip side, you can set the maximum size using:

```
MAXWINDOW 500,400
```

Again, this is a global setting in the runtime. If different windows need different settings, just reset the minimum or maximum size each time the window is made to the output, or frontmost window.

Is Window There?

One way to determine if a window has already been built, is to use the WINDOW function. When given a -wndID%, it returns zero if the window has not already been built, and a value if it has (actually, the window pointer). Thus, to see if window #15 is already present, yet possibly hidden, do this:

```
LONG IF WINDOW (-15)
' window #15 is already built
XELSE
' window #15 is not built, so build it here
END IF
```

So, to determine if a window is already built, use the WINDOW function like this:

```
IsWindowBuilt = WINDOW (-wndID%)
```

Window Output

With so many windows capable of appearing on the screen, there must be some method of designating which window will receive the text and graphics you want to display. Fortunately, there are two ways of accomplishing this. One uses the WINDOW statement, the other uses the WINDOW OUTPUT statement.

The WINDOW statement accepts a wndID% and makes that window the frontmost one on the screen. It redirects text and graphics commands to that window. For example, to direct output to window #15, just do this:

```
WINDOW #15
```

In contrast, WINDOW OUTPUT also accepts a wndID%, but it directs any subsequent text and graphics commands to the specified window without making it the frontmost window. This is handy for updating windows that are behind others on the screen. If window #15 is the frontmost window, but window #7 needs updating behind it, use:

```
WINDOW OUTPUT #7
' -- update window #7 here
WINDOW OUTPUT #15
```

to direct output to window #7, update its contents, then redirects output back to window #15. All this is done without changing the order of the windows on the screen.

Closing Windows

Use `WINDOW CLOSE` with the appropriate `wndID%` to dispose of the window and all its associated items (buttons, edit and picture fields).

If that's all there is to closing a window, why make such a big deal about it? Because, it isn't quite that easy (sigh!). Re-read the second line above, especially the part about "all its associated items...". That's the trick. We must make sure we have gathered all the data from a window's buttons and fields *before* it's disposed of. If we don't, we have no way of recovering it. We'll see how to handle that later on in the chapter.

Regular Exercise

Now that we understand windows better, we can begin adding them to *SimpleBase*. We'll follow almost the same steps that we used to create our menus. Look at Table 4 on the next page to see the program windows and a brief description what each one does.

Window Routines

There are three main window handling routines that every program should have: building, capturing a window's data, and closing the window. Each performs a specific task related to window management including. Let's look at each one briefly before we dive into the code that makes them work.

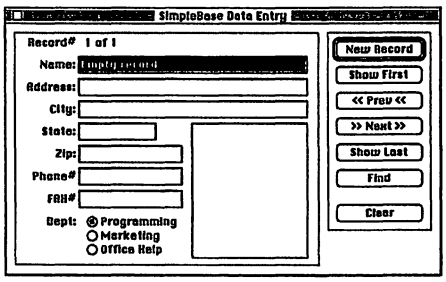
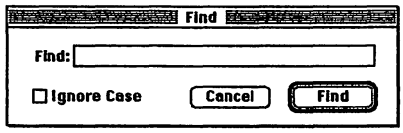
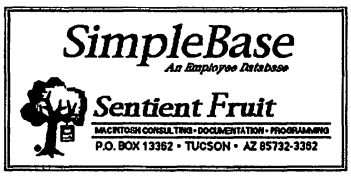
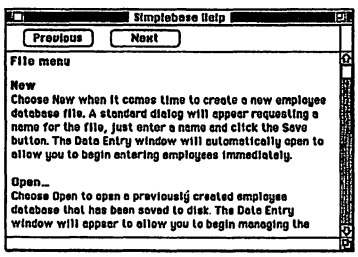
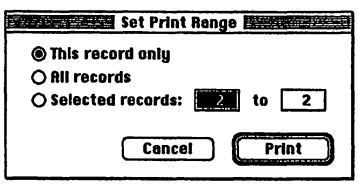
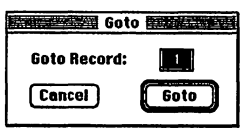
Window Build Routine

The window building routine is straightforward. We need a short routine that will build any window and then display it. It should also know if a window has already been built and just display it, instead of rebuilding the entire window (a slow and painful process for the user). The pseudocode to accomplish these goals is shown in Program 22.

PROGRAM 22. Build window pseudocode.

```
Determine if window needs building
   If not built, determine which window to build
   build correct window
when window is built, just show it
```

TABLE 4. SimpleBase window descriptions.

PROGRAM WINDOW	DESCRIPTION
	*Data Entry* — the main database entry window. Here the user can maneuver around the current open database, add new records, clear current ones, and edit record data. Record data is automatically saved as the user moves through the file.
	*Find* — this window appears when the user selects **Find** from the **Records** menu or clicks the **Find** button in the data entry window. Here the text is entered for the search in the current active database.
	*About* — this window (really an alert) appears when the user selects the **About SimpleBase** item from the menu. It provides information on the program.
	*Help* — appears when the user selects **Help** from the menu. It contains a scrolling listing of help text. The user clicks on the buttons to move through the various help screens.
	*Print* — appears when the user tries to print something. It allows them to print a single record, any subset of records, as well as all records.
	*Goto* — appears when the user chooses **Goto** from the **Records** menu. Enables them to directly jump to a specific record in the database.

Window Capture Routine

The routine to capture window data is not as simple. `WindowCapture` is called when a window is about to be closed that contains data the user wouldn't like to lose, or just to read the data in the window. Since a window's edit fields and controls are destroyed when the window is closed, the data contained would be irretrievable. This routine ensures that the user loses no data. The pseudocode for the `WindowCapture` subroutine is shown in Program 22.

PROGRAM 23. Capture window pseudocode.

```
Determine the window to capture information from
Does any data in the window require saving
Save the data
```

Window Close Routine

The window closing routine is also pretty straightforward. It closes the specified window after calling the window capture routine and after disposing of anything in the window that needs it. While we have no structures in *SimpleBase* that illustrate this technique, the explanation is simple.

Here, in Program 24, are the three skeleton window routines required by *SimpleBase* to capture, close, and build windows. Each accepts a `wndID%` parameter to specify which window they should operate on.

PROGRAM 24. Skeleton window routines.

```
LOCAL FN WindowCapture (wndID%)
END FN
LOCAL FN WindowClose (wndID%)
END FN
LOCAL FN WindowBuild (wndID%)
END FN
```

Now that we have our window handling routines ready to go, let's start building some windows.

Window Constants

Let's continue by defining our program's window constants. We define them as shown in Program 25, creating one for each window as previously described in Table 4.

PROGRAM 25. Window constant definitions.

```
' >>> WINDOWS
_dbEntryWIND    = 1
_dbFindWIND     = 2
_aboutWIND      = 3
_helpWIND       = 4
_printWIND      = 5
_gotoWIND       = 6
```

Building the Windows

Building our program windows falls directly on the subroutine `WindowBuild`. It's whole purpose in life is to direct the building of program windows. We already saw the pseudocode, but let's look at it formatted as a `LOCAL FN` in Program 26.

PROGRAM 26. Build window function in pseudocode

```
LOCAL FN WindowBuild (wndID%)
   Determine if window needs building
   If not built, determine which window to build
   call subroutine to build window
   when window is built, just show it
END FN
```

Translating the rest of the pseudocode into equivalent BASIC statements is quick and easy. Since the pseudocode describes a choice between two actions, that choice indicates the use of a branch block. In this case, let's use a `LONG IF` structure.

Earlier we mentioned that our `WindowBuild` routine should be smart enough to know if a window has already been built. We know that a version of the `WINDOW` function can do this using a negative `wndID%`, so we add that to the `LONG IF` test. If the window is already built, just redisplay it using the `WINDOW` statement, if not, we have to build it first.

Next, in the `SELECT` portion of `WindowBuild`, we can add a `CASE` statement for each window defined earlier, and add the call to each window's build routine. The final `WindowBuild` routine can be seen in Program 27.

PROGRAM 27. Expanded WindowBuild routine

```
LOCAL FN WindowBuild (wndID%)
  LONG IF WINDOW (-wndID%) = 0
    SELECT wndID%
      CASE _dbEntryWIND : FN BuildEntryWnd
      CASE _dbFindWIND : FN BuildFindWnd
      CASE _aboutWIND : FN BuildAboutWnd
      CASE _helpWIND : FN BuildHelpWnd
      CASE _printWIND : FN BuildPrintWnd
      CASE _gotoWIND : FN BuildGotoWnd
    END SELECT
  END IF
  WINDOW #wndID%
END FN
```

Window Building Routines

In the spirit of top-down design, we shuffle the actual window building to other subroutines. In our case, six more routines do the low level work of creating the windows. Each routine performs essentially the same task: it builds the window along with the window's controls, edit fields, and picture fields. Let's look closely at one of these building routines.

The main window is, of course, the Data Entry window. This is where users will be able to enter their employee data. With all that information, the Data Entry window will be the most complex of our windows. However, since each follows the same general pattern, no window is any harder to build than another. The pseudocode shown in Program 28 is the skeleton code for all of our window building subroutines.

PROGRAM 28. Build window skeleton.

```
LOCAL FN BuildWindowSkeleton
  create window as required
  assign the font, size, style, and mode
  build all window buttons
  build all window edit fields
  assign any preset conditions (to buttons, fields, etc)
END FN
```

Why do we add all the controls and fields in the build window subroutines? Because of a simple fact, the runtime updates them all for us. That means that once we've built a window, and added the appropriate controls and fields to it, we never have to worry about building them again. Unless we change a

name, change an item's location, or close a button or field, we don't have to worry about them.

We'll see how to add buttons and edit fields in later chapters but now let's look at the construction of the window. To build the Data Entry window, examine the routine shown in Program 29.

PROGRAM 29. Build Data Entry window.

```
LOCAL FN BuildDataEntryWnd
  tmp$ = "SimpleBase Data Entry"
  WINDOW #-_dbEntryWIND, tmp$, (0,0)-(500,300), _docNoGrow, _dbEntryWIND
  TEXT _sysFont, 12
  ' add buttons
  ' add edit/picture fields
  ' assign preset values
END FN
```

We start by assigning a window title to a temporary string. This may seem silly to do at this point, but later when we move all of our string data to STR# resources in "Creating Program STR#", it will make more sense.

Next, we use the WINDOW statement to build the window. Here we pass it the window ID (as one of our predefined constants), the title in our temporary string, its location on the screen (centered, of course), its window type and window class (identical to our defined constant).

Once the window has been built invisibly by its build routine, a second WINDOW statement makes the window visible just before exiting the WindowBuild subroutine.

The next line uses the TEXT statement to assign a default font and font size to the new window. FB builds a window with a default font and size, which may not be what you want. In this case I wanted Chicago 12 as my window font. This is the font and size FB will use when creating edit fields or for printing in the window.

Using the same general design, it's now possible to write all of our window building routines. You can see how they came out by examining the code in Program 30.

PROGRAM 30. Window building routines.

```
LOCAL FN BuildDataEntryWindow
  tmp$ = "SimpleBase Data Entry"
  WINDOW #-_dbEntryWIND, tmp$, (0,0)-(500,290), _docNoGrow, _dbEntryWIND
  TEXT _sysFont, 12
END FN
LOCAL FN BuildFindWindow
  tmp$ = "SimpleBase Find"
  WINDOW #-_dbFindWIND, tmp$, (0,0)-(340,80), _docNoGrow, _dbFindWIND
  TEXT _sysFont, 12
END FN
LOCAL FN BuildAboutWindow
  ' will use alert - see Alerts chapter
END FN
LOCAL FN BuildHelpWindow
  tmp$ = "SimpleBase Help"
  WINDOW #-_helpWIND, tmp$, (0,0)-(400,260), _docZoom, _helpWIND
  TEXT _sysFont, 12
END FN
LOCAL FN BuildPrintWindow
  tmp$ = "Print Record"
  WINDOW #-_printWIND, tmp$, (0,0)-(300,125), _docNoGrow, _printWIND
  TEXT _sysFont, 12
END FN
LOCAL FN BuildGotoWindow
  tmp$ = "Goto Record"
  WINDOW #-_gotoWIND, tmp$, (0,0)-(200,80), _docNoGrow, _gotoWIND
  TEXT _sysFont, 12
END FN
```

Window Testing

With our window building routines finished, you may notice that they aren't accessible to the rest of the program as it is now. We can, however, add some additional lines of code to display each window using a menu command.

This is important. Remember the spirit of stepwise refinement states that at each stage of the design process we test the design to ensure it works. In this case, we need to check each window to see that it's the correct window type, has the correct size, and most of all, that the `WindowBuild` routine, and window build subroutines all work as designed.

For example, the logical place to test build our data entry window is under the **New** item on the **File** menu. Opening the Print window should be done under the Print item of the File menu. For the Find and Goto windows, it's **Find...**

PROGRAM 31. Window building calls.

```
LOCAL FN ItemHelp
   FN WindowBuild (_helpWIND)
END FN
LOCAL FN ItemNew
   FN WindowBuild (_dbEntryWIND)
END FN
LOCAL FN ItemFindRecord
   FN WindowBuild (_dbFindWIND)
END FN
LOCAL FN ItemPrint
   FN WindowBuild (_printWIND)
END FN
LOCAL FN ItemGotoRecord
   FN WindowBuild (_gotoWIND)
END FN
```

and **Goto...** under the **Records** menu. The Help window should be built from the menu. To achieve this, add the lines of code shown in Program 31 to the specified subroutines.

Run *SimpleBase*. You can now display any of our six program windows by selecting, **Help**, **New**, **Print**, **Find...**, or **Goto...** from the appropriate menu. Unfortunately, you can't close any of them just yet, so let's look into that.

Closing Windows

As previously mentioned, closing a window is simple, however, it's important to preserve any information stored in the buttons and edit fields within the window before it's closed. Otherwise, that information will be lost forever. This routine can be seen in Program 32.

PROGRAM 32. Close window test.

```
LOCAL FN ItemClose
  FN WindowClose (WINDOW (_outputWnd))
END FN
```

We make sure the data is saved by calling the `WindowCapture` routine before closing the window. This guarantees that our information will be saved whenever the window is closed. We also include a `SELECT/END SELECT` structure in there, in case there are any special features in the window that

PROGRAM 33. Simple window close routine.

```
LOCAL FN WindowClose (wndID%)
  LONG IF FN WindowCapture (wndID%)
    SELECT wndID%
      CASE _dbEntryWIND
      CASE _dbFindWIND
      CASE _aboutWIND
      CASE _helpWIND
      CASE _printWIND
      CASE _gotoWIND
    END SELECT
    WINDOW CLOSE #wndID%
  END IF
END FN
```

must be disposed of prior to calling `WINDOW CLOSE`. Examine the listing in Program 33 to see a simple close window routine.

We pass `WindowClose` the `wndID%` of the current frontmost window and let it take care of the rest. We will also add more calls to `WindowClose` as they become necessary.

Capturing Window Data

The `WindowClose` routine provides a handy spot to determine if a window that's about to be closed has any worthwhile data to save. If it does, we call the `WindowCapture` routine shown in Program 34 to handle all the sundry details of saving our window's data. Here we set a `closeFlag%` variable upon entry which in turn is passed back to the calling function. If, for any reason the close should need to be canceled, just zero out the `closeFlag%` and prevent the window from being closed.

At this point, we don't really have anything to save but it's important to make these decisions now. We'll see how to implement saving later when we have some buttons and fields in the window to work with.

That covers everything on windows needed to get them up and running on *SimpleBase*. Feel free to skip ahead to the "Buttons" chapter where we start filling in windows with usable controls.

PROGRAM 34. Simple window capture routine.

```
LOCAL FN WindowCapture (wndID%)
  SELECT wndID%
    CASE _dbEntryWIND
    CASE _dbFindWIND
    CASE _aboutWIND
    CASE _helpWIND
    CASE _printWIND
    CASE _gotoWIND
  END SELECT
END FN
```

Peak Performance

The following are useful window subroutines that show some of the interesting things you can do with a window once it's been built.

Window Record

A Macintosh window is stored in memory as a window record. It contains everything you'd ever want to know about a particular window. We won't detail the window record here since its structure can be found in the *Reference* manual under `GET WINDOW` and in *Inside Macintosh: Macintosh Toolbox Essentials.*

The key to accessing all this window information depends on getting a pointer (a memory address) to the record. You can get this pointer using one of two methods: the `GET WINDOW` statement or the `WINDOW(_wndPointer)` function. Once you have a valid pointer to a window record you can examine, modify, or retrieve information contained within.

Window Titles

Sometimes a program needs to know which window is currently the frontmost by name instead of window number or class. The following Toolbox procedure `GetWTitle` demonstrates how to retrieve a window name using the `wndID%`:

```
CALL GETWTITLE (WINDOW (_wndPointer), wndTitle$)
PRINT wndTitle$
```

And, since you may sometimes need to change a window's title, use the Toolbox procedure `SetWTitle` to replace the current title with a new one:

```
wndTitle$ = "Database Listing"
CALL SETWTITLE (WINDOW (_wndPointer), wndTitle$)
```

Resizing Windows

Often it may be necessary to resize a window, either to display more information to the user by making it larger, or to hide information by making it smaller. The example in Program 33 shows how to use the `SizeWindow` procedure to do both.

PROGRAM 35. Resizing window example.

```
LOCAL FN ResizeWindow (wndID%, newX%, newY%)
  oldWndID% = WINDOW (_outputWnd)
  WINDOW OUTPUT wndID%
  GET WINDOW wndID%, wndPtr&
  CALL SIZEWINDOW (wndPtr&, newX%, newY%, _zTrue)
  WINDOW OUTPUT oldWndID%
END FN
```

Cooldown

In this chapter you learned all about windows, what they are, their various features, the different window types and their major uses in a program. Along the way we talked about centering windows on larger screens and building windows invisibly to give the illusion of greater speed when they first appear to the user.

Additionally, we described the six windows used by *SimpleBase*, and walked you through creating three window handling subroutines (build, close, and capture) that every program will use. Also, we described in both pseudocode and keywords how to quickly implement these routines in a program.

With our windows built, it's now time to add some functionality to them. We start in the next chapter by adding buttons.

CHAPTER 6

Buttons

Warm-up

This chapter introduces you to controls, more commonly referred to in FB as buttons. In this chapter you will:

- Learn what buttons are,
- Identify the four button types,
- Learn how to create buttons, and
- Learn how to handle specific button actions.

What are Buttons?

Buttons, or controls, are selectable objects in a window that maintain a value or perform some type of action. The purpose of a button is normally indicated by a title, but can be represented by icons or pictures.

Buttons are redrawn automatically by the FB runtime package. That means a program can create them once in a window and never worry about them again.

Creating Buttons

Buttons are identified by a `btnID%`. Each window in a program can contain up to 8192 buttons. Assuming you had enough memory to hold them all, that comes to a staggering 516,096 buttons for 63 program windows. Each window maintains its own internal list of buttons. That means that window

#1 can have a button #1, window #2 can have a button #1, and so on. Note, that FB doesn't allow multiple buttons with the same ID within one window.

You can create any of the four button types using the BUTTON statement. BUTTON requires five pieces of information, a btnID% to uniquely identify the button in the window, a button state, a title, a location in the window, and a button type. For example, to create any button in a window you would do this:

```
BUTTON #btnID%, btnState%, title$, (left, top)-(right, bottom), btnType%
```

Of course, the btnID% is used extensively to identify the button, both on creation, and later when we need it to determine which button was selected in a window.

Button States

Every button can have one of three states as shown in Table 5. Your program can control a button's appearance by resetting the btnState% parameter in the BUTTON statement.

Just as with menus, graying a button enables the program to guide the user's choices. It disables button choices that shouldn't be available and enables button choices that are.

TABLE 5. Button states.

STATE	DESCRIPTION
_grayBtn	*Inactive button.* User can't select.
_activeBtn	*Active button.* User can select.
_markedBtn	*Selected button.* Active and selected by the user.

Button Titles

A button title can be any string up to 255 characters. For push buttons and shadow buttons, titles are normally restricted to just a few words and preferably just one. Most push buttons have names like **Done**, **Cancel**, and **Save**. Not very exciting as titles go, but they get the job done.

Checkboxes and radio buttons can, and should, have titles that describe the button's purpose completely. Titles like: "Enable the turbo-warp drive on start-up" describe exactly what the user is setting the button for.

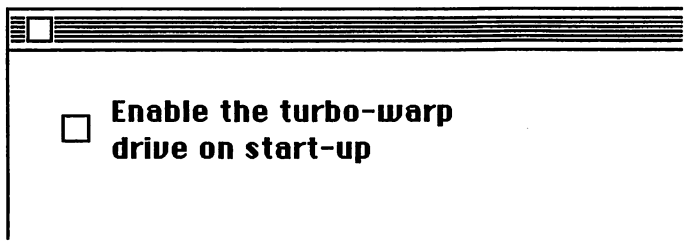
It's possible to wrap a long title so that the button displays its title on two or more lines. While not common, it's sometimes necessary. To do this, insert a carriage return into the title where the break should occur. Naturally, you

must adjust the button's height to accommodate this extra line. For example. to display the long title in the previous paragraph you can define it like this:

```
title$ = "Enable the turbo-warp" + CHR$(13) + "drive on start-up"
BUTTON #1, _activeBtn, title$, (10, 10) - (100, 40), _checkbox
```

The use of multiple title lines should only be used on buttons as a last resort when a shorter title will not work.

FIGURE 24. Forcing multiple lines in buttons.

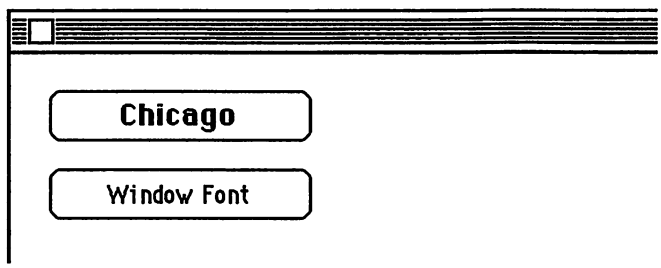


Most buttons use the Chicago font to display their titles. In some cases, you may need to use an alternate font for a title display. To do so, you must append the `_useWFont` setting to the button type. This forces the runtime to use the current window font as its display font. For example, it's easy to mix and match fonts in a window like this:

```
BUTTON #1, activeBtn, "Chicago", (10, 10)-(100, 25), _push
BUTTON #2, activeBtn, "Window Font", (10, 40)-(100, 55), _push_useWFont
```

However, it's not recommended. Don't confuse a user by displaying your mastery of font juggling in button titles. Simple, direct, and readable fonts like Chicago work just fine, without any embellishment.

FIGURE 25. Buttons using Chicago and window fonts.



Button Positioning & Sizes

The coordinates of a button are always defined in relation to the top-left corner of the window's content area, not the title bar. Button coordinates are used by the button to determine one thing: the area where button clicks will be detected.

Buttons should be wide enough to display the entire title. Radio buttons and checkboxes can be very wide if the title is descriptive. Push and shadow buttons should leave enough room at each end so that the titles don't appear cramped inside them.

Buttons should also have enough height to clearly present the title. For example, the default height for a push button is 20 pixels, for checkboxes and radio buttons it's normally 15 pixels.

- *Note, for SimpleBase we use FB's default* `COORDINATE WINDOW` *setting. This causes our coordinates to match the screen resolution of 72 dpi. If you use* `COORDINATE x, y` *to define any other coordinate system, you are on your own calculating where items will appear in a window.*

Closing Buttons

There are two ways to close a button. The first works upon an individual button using the `BUTTON CLOSE` statement. Just pass it a `btnID%` value and the button closes. The second method is to simply close the window containing the button. If you have a lot of buttons to close, this is probably the best method to ensure all are closed properly.

Button Types

The final parameter identifies the button type to create using one of FB's pre-defined constants. There are several types of buttons including: push button, checkbox, radio, and a variation of the push button, the shadow button. Each performs a specific task when clicked by the user.

TABLE 6. Button types

BUTTON	DESCRIPTION
Push	*Push* — when clicked, a program should perform the action defined by the button title immediately. Push buttons should be large enough to hold the title.
☒ On ☐ Off ☒ Disabled	*Checkbox* — clicking a checkbox button allows the user to enable or disable the option specified by the button text.
◉ Choice 1 ○ Choice 2 ○ Choice 3	*Radio* — provide between two and seven mutually exclusive choices. In each group of radio buttons, only one can ever be on at a time.
Shadow	*Shadow* — usually the default button in a window. Users can click with the mouse or press the Return or Enter key to activate a shadow button.

Regular Exercise

Now that we understand more about buttons, let's begin adding them to our program windows. We start by creating a few button constants that will be used by the various windows in *SimpleBase*.

Again, as was done with windows, I end each button constant with the suffix `BTN` (or `SCROLL` for scroll buttons). That way I never have to think about what the constant refers to. The complete list of constants is shown in Program 36. The constants are defined, now it's time to create the buttons.

Creating Program Buttons

Before we get started, let's state one truism about them:

Buttons, edit, and picture fields are automatically redrawn by the runtime.

That means that once we've defined a button (or edit or picture field) in a window, we never have to go back and redraw it for any reason. The runtime

PROGRAM 36. Button constants.

```
' >>> BUTTONS
' >>> DATA ENTRY WINDOW
_doneBTN        = 1
_newRecBTN      = 2
_firstRecBTN    = 3
_prevRecBTN     = 4
_nextRecBTN     = 5
_lastRecBTN     = 6
_deleteRecBTN   = 7
_programBTN     = 8
_marketBTN      = 9
_officeBTN      = 10
' >>> FIND WINDOW
_findBTN        = 1
_cancelBTN      = 2
_ignoreCaseBTN = 3
' >>> ABOUT WINDOW
_okBTN          = 1
' >>> HELP WINDOW
_helpSCROLL     = 1
' >>> PRINT WINDOW
_printBTN       = 1
_thisRecBTN     = 3
_allRecBTN      = 4
_selectRecBTN   = 5
' >>> GOTO WINDOW
_gotoBTN        = 1
```

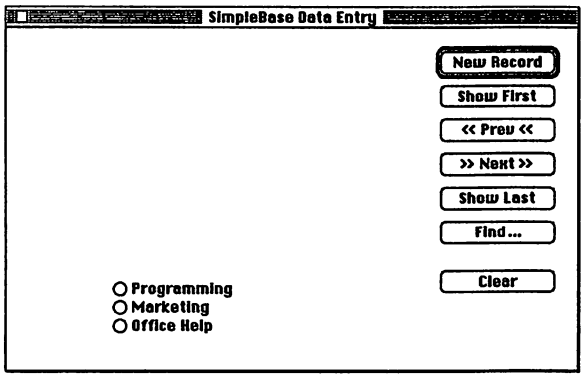
takes care of that for us. We may change its title, state, or location in the window, but we never have to worry about refreshing it.

What else have we already built that doesn't need to be refreshed? If you answered "window", you're absolutely correct. Therefore we can bundle our button creation routines within our window building routines. Let's start with the most challenging window, the data entry window.

Data Entry Window Buttons

Most of the program's activity will take place in the main Data Entry window, so it's vital that we offer as many options as possible without overpowering either the window, or the user with too many buttons. The Data Entry window with its numerous buttons is shown in Figure 26.

FIGURE 26. Buttons in Data Entry window.



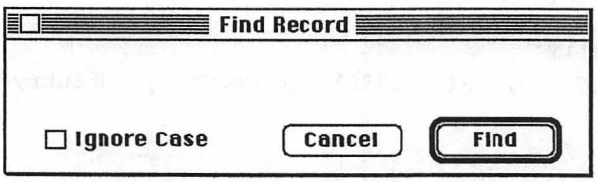
PROGRAM 37. Adding buttons to BuildEntryWindow.

```
LOCAL FN BuildEntryWindow
  tmp$ = "SimpleBase Data Entry"
  WINDOW #-_dbEntryWIND, tmp$, (0,0)-(500,290), _docNoGrow, _dbEntryWIND
  TEXT _sysFont, 12, ,0
  ' ••• BUTTONS
  tmp$ = "New Record"
  BUTTON _newRecBTN,_activeBtn,tmp$,(380,20)-(480,40), _shadow
  tmp$ = "Show First"
  BUTTON _firstRecBTN,_activeBtn,tmp$,(380,50)-(480,70),_push
  tmp$ = "<< Prev <<"
  BUTTON _prevRecBTN,_activeBtn,tmp$,(380,80)-(480,100),_push
  tmp$ = ">> Next >>"
  BUTTON _nextRecBTN,_activeBtn,tmp$,(380,110)-(480,130),_push
  tmp$ = "Show Last"
  BUTTON _lastRecBTN,_activeBtn,tmp$,(380,140)-(480,160),_push
  tmp$ = "Find"
  BUTTON _findRecBTN,_activeBtn,tmp$,(380,170)-(480,190),_push
  tmp$ = "Delete"
  BUTTON _deleteRecBTN,_activeBtn,tmp$,(380,210)-(480,230), _push
  tmp$ = "Programming"
  BUTTON _programBTN,_activeBtn,tmp$,(90,222)-(200,237),_radio
  tmp$ = "Marketing"
  BUTTON _marketBTN,_activeBtn,tmp$,(90,238)-(200,253),_radio
  tmp$ = "Office Help"
  BUTTON _officeBTN,_activeBtn,tmp$,(90,254)-(200,269),_radio
  ' ••• EDIT/PICTURE FIELDS
END FN
```

Since we've already defined our button constants, let's add them to the `BuildEntryWindow` subroutine with the `BUTTON` statement. Each statement sets the button title in a string variable, sets its initial state, defines its position in the window, and set the button type. You can view the changes made to `BuildEntryWindow` in Program 37.

The first seven buttons are push buttons that will execute commands useful to the user while in the data entry window. They allow the user to add new records, find records, clear records, as well as maneuver forward and backward in the database. Once the buttons are added, we can view their positions by running the program and opening the Data Entry window using **New** from the **File** menu. Ideally, you should see something that looks a lot like Figure 26.

FIGURE 27. Find window button positions.



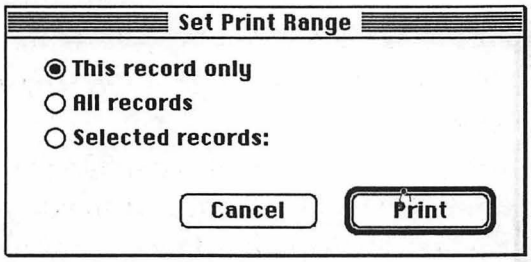
Find Window Buttons

The Find window doesn't have quite the same complexity as does the Data Entry window. Its three buttons make for a much more compact build routine. The layout of the Find window with its three buttons can be seen in Figure 27, while the code to add them is shown in Program 38.

PROGRAM 38. Find window buttons.

```
LOCAL FN BuildFindWindow
   tmp$ = "Find Record"
   WINDOW #-_dbFindWIND, tmp$, (0,0)-(340,80), _docNoGrow, _dbFindWIND
   TEXT _sysFont, 12
   ' ••• BUTTONS
   tmp$ = "Find"
   BUTTON _findBTN,_activeBtn,tmp$,(250,50)-(320,70),_shadow
   tmp$ = "Cancel"
   BUTTON _cancelBTN,_activeBtn,tmp$,(160,50)-(230,70),_push
   tmp$ = "Ignore Case"
   BUTTON _ignoreCaseBTN,_activeBtn,tmp$,(20,50)-(150,70), _checkBox
END FN
```

FIGURE 28. Print window button positions.



Print Window Buttons

The next buttons to create belong to the Print window. Again, we have a default **Print** button, a **Cancel** button, and three grouped radio buttons defining which records will be printed. We'll see how to handle these radio buttons as a group quickly later in this chapter. The Print window layout is shown in Figure 28 while the code to create it is shown in Program 40.

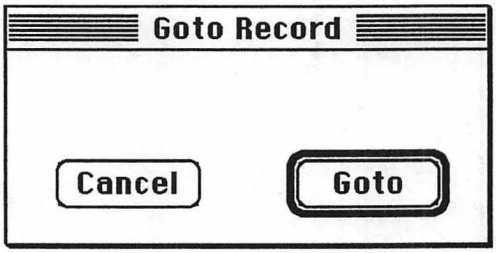
PROGRAM 39. Print window buttons.

```
LOCAL FN BuildPrintWindow
  WINDOW #-_printWIND, "",(0,0)-(300,140),_docNoGrow, _printWIND
  TEXT _sysFont, 12, , 0
  ' ••• BUTTONS
  tmp$ = "Print"
  BUTTON _printBTN,_activeBtn,tmp$,(200,90)-(280,110), _shadow
  tmp$ = "Cancel"
  BUTTON _cancelBTN,_activeBtn,tmp$,(100,90)-(180,110), _push
  tmp$ = "This record only"
  BUTTON _thisRecBTN,_activeBtn,tmp$,(20,10)-(200,25), _radio
  tmp$ = "All records"
  BUTTON _AllRecBTN,_activeBtn,tmp$,(20,30)-(200,45), _radio
  tmp$ = "Selected records:"
  BUTTON _selectRecBTN,_activeBtn,tmp$,(20,50)-(160,65), _radio
END FN
```

Goto Window Buttons

The Goto window is simplicity itself. There are only two buttons available here, a default **Goto** and a **Cancel** button. The window layout is shown in Figure 29 while the code to create this window is shown in Program 40.

FIGURE 29. Goto window button positions.



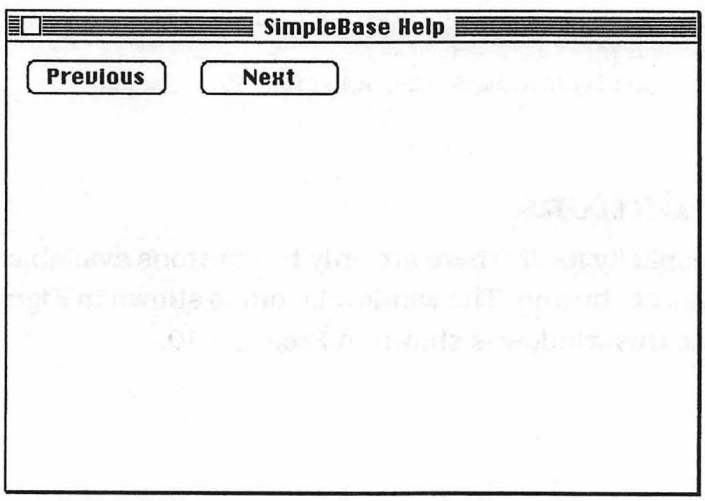
PROGRAM 40. Goto window buttons.

```
LOCAL FN BuildGotoWindow
  WINDOW #-_gotoWIND, "",(0,0)-(200,80),_docNoGrow, _aboutWIND
  TEXT _sysFont, 12, , 0
  ' ••• BUTTONS
  tmp$ = "Goto"
  BUTTON _gotoBTN,_activeBtn,tmp$,(120,45)-(180,65)), _shadow
  tmp$ = "Cancel"
  BUTTON _cancelBTN,_activeBtn,tmp$,(20,45)-(80,65)), _push
END FN
```

Help Window Buttons

The final window to receive buttons is the Help window. Here we have two buttons, **Previous** and **Next**, that allow the user to cycle through the various help messages. The layout for this window's buttons is shown in Figure 30 while the code to create this window is shown in Program 40.

FIGURE 30. Help window buttons.



PROGRAM 41. Help window buttons.

```
LOCAL FN BuildHelpWindow
  tmp$ = "Simplebase Help"
  WINDOW #-_helpWIND, tmp$, (0,0)-(400,260), _docZoom, _helpWIND
  TEXT _sysFont, 12
  ' ••• BUTTONS
  tmp$ = "Previous"
  BUTTON #_prevHelpBTN, 1, tmp$, (20,5)-(100,25), _push
  tmp$ = "Next"
  BUTTON #_nextHelpBTN, 1, tmp$, (120,5)-(200,25), _push
END FN
```

Handling Button Actions

Once again, we are back to dealing with events. As stated before, the program should never anticipate the arrival of an event, instead it should wait until it receives an event before responding. This is especially true with buttons.

The key to handling buttons is to get the event in the Main Loop, and if it's a button event then pass it onto a dialog handling subroutine. It in turn determines the type of event and passes it to the appropriate window's dialog handling subroutine. Just as we did with building our windows in the last chapter, we now create a dialog handling subroutine for each window in the program to deal with its own dialog events. We'll see exactly how to do this in the next chapter "Dialog Events".

Before we do that, let's briefly look at some standard methods of dealing with different button events no matter which window they reside in.

The `BUTTON` function returns the value of the specified button. We can use this in many of the following generic subroutines to determine program logic based on the current state of the button. For example, to return the state of a button:

```
btnState% = BUTTON (btnID%)
```

Push Buttons

Handling a button event for a push button is simple. The program determines which button was pressed and immediately responds accordingly. In *SimpleBase*, push buttons will open the Find window, control movement among our records, close windows, and set program and record options.

Push buttons only use two of the three states: `_activeBtn`, and `_grayBtn`. They only briefly use the `_markedBtn` setting when highlighted by a mouse click.

Checkboxes

Handling checkboxes is a little more complicated than push buttons, but isn't really hard. Checkboxes have two active states (_activeBtn and _markedBtn), alternating between them at user selection. The routine to handle this for any program's checkbox is shown in Program 42.

We use the BUTTON function to determine the state of the specified button. If the button is in the marked state, we unmark it, otherwise, we mark it. Additionally, the routine returns the current button state so that the user's choices can be stored in a preferences file.

PROGRAM 42. Handling checkboxes.

```
LOCAL FN CheckBoxHandler (btnID%)
  LONG IF BUTTON (btnID%) = _markedBtn
    BUTTON btnID, _activeBtn
  XELSE
    BUTTON btnID, _markedBtn
  END IF
  btnState% = BUTTON (btnID%)
END FN = btnState
```

This is one for your own library of useful routines. Build it once and never worry about it again.

Radio Buttons

Handling radio buttons is a common interface hurdle faced by Macintosh programmers. How do you ensure that only one radio button of any group is active? Briefly, since they are clustered together into groups, a subroutine can cycle through all of the buttons in the group to turn on the correct one while making sure the rest are off. The only requirement for this routine in Program 43 is that you define each grouping of radio buttons in sequence.

PROGRAM 43. Handling radio buttons.

```
LOCAL FN RadioButtonHandler (firstBtn%, lastBtn%, btnID%)
  FOR count% = firstBtn% TO lastBtn%
    LONG IF count% = btnID%
      BUTTON count%, _markedBtn
    XELSE
      BUTTON count%, _activeBtn
    END IF
  NEXT count%
END FN = btnID%
```

PROGRAM 44. HiliteSelectedButton routine.

```
LOCAL FN HiliteSelectedButton (btnID%)
  BUTTON btnID%, _markedBtn
  DELAY _secTick
  BUTTON btnID%, _enableBtn
END FN
```

The routine uses a FOR/NEXT loop that begins with the first radio button (firstBtn%) in the group and cycles through to the last (lastBtn%). It sets each radio button to _activeBtn as it goes, ensuring that only one radio button per group is ever marked at the same time. When the button specified by btnID% is found, the LONG IF is executed and the button state is changed to _markedBtn.

This is another routine to add to your own library of useful subroutines.

Shadow Buttons

Shadow buttons are push buttons that have a thick rectangle around them that identifies them as the default button in the window. Other than the thick outline, shadow buttons are identical to regular push buttons.

While a shadow button is nothing more than a fancy push button, a program should treat it a little differently than a regular push button. Specifically, when the user hits the Return or Enter key, the default button in the window is expected to respond just as if the mouse had clicked on it. To get that kind of behavior, use the routine shown in Program 44.

When passed a btnID%, the HiliteSelectedButton subroutine sets the specified button to marked. In this case, the push button inverts, just as if it were clicked with the mouse. A slight pause then allows the user to see the inverted button, then it's reset to the normal active state.

Enabling & Disabling Buttons

As previously mentioned, it's possible to disable, or make unselectable, any button that shouldn't be available to the user. The function shown in Program 45 looks at the current state of the button and switches it from inactive to enabled.

While this routine is simple, it's often useful and should be stored in your subroutine library.

PROGRAM 45. Enable button function.

```
LOCAL FN EnableButton (btnID%)
  LONG IF BUTTON (btnID%) = _grayBtn
    BUTTON btnID%, _activeBtn
  XELSE
    BUTTON btnID%, _grayBtn
  END IF
END FN
```

Peak Performance

The use of FB's four button types will satisfy the majority of programmer requirements. However, there may come a time when you need to do something slightly different with a button. For that, you need access to the control (or button) record.

Every button you build in a window has an internal record that contains all the information pertaining to that button. I won't detail the control record here since it is described completely in the FB *Reference* manual under BUTTON& and in *Inside Macintosh: Macintosh Toolbox Essentials.*

Button Handles

The location of a control record is stored in memory as a handle. A handle is an address that points to another address, which itself points to the actual control data. With a handle to a control record, it's possible to manipulate the control in ways not directly available in FB statements.

- *For more information on handles and pointers see the chapter "Resources".*

You can get a handle to a control using the BUTTON& function. When given a btnID%, BUTTON& returns a handle to that button's control record. For example, to get the handle to the **Find** button in the Find window of *SimpleBase*, the window must first be opened. Then do this:

```
findBtnH& = BUTTON& (_findBTN)
```

That's all there is to it.

Getting & Setting Button Titles

Another thing you can do with a control handle is retrieve or set the title of a control. For example, to get a button's title use the Toolbox GetCTitle procedure like this:

```
CALL GETCTITLE (BUTTON& (btnID%), ctlTitle$)
PRINT ctlTitle$
```

To replace a button title use the Toolbox procedure `SetCTitle`:

```
ctlTitle$ = "Reset"
CALL SETCTITLE (BUTTON& (btnID%), ctlTitle$)
```

This is a much better method than rebuilding the entire button each time its name changes.

Hiding Buttons

Occasionally, it may be necessary to hide a button in a window. A routine that provides this capability is shown in Program 46. It makes use of two Toolbox procedures, `HideControl` and `ShowControl`, and a flag variable to specify the action it should perform.

PROGRAM 46. Hide and show button function.

```
LOCAL FN HideShowBtn (btnID%, hideFlag%)
  cntrlH& = BUTTON&(btnID%)
  LONG IF hideFlag%
    CALL HIDECONTROL (cntrlH&)
  XELSE
    CALL SHOWCONTROL (cntrlH&)
  END IF
END FN
```

Cooldown

In this chapter we've introduced you to buttons. Along the way you learned the four different button types and their appropriate uses in programs. We created buttons in our project and learned some useful subroutines for handling specific button actions for checkboxes, radio buttons, and shadow buttons.

In the next chapter we'll learn how to handle these buttons as well as window events so that *SimpleBase* can begin acting like a real program.

Dialog Events

Warm-up

The efficient handling of dialog events is critical to the smooth and seamless operation of an application. This chapter continues your education in handling events, especially dialog events. In this chapter you will:

- Learn what dialog events are,
- Identify the window an event belongs to, and
- Write routines to handle window, button, and cursor events.

What are Dialog Events?

Dialog events are events generated by the operating system that are processed by the FB runtime. Unlike mouse and menu events, dialog events are generated by many Macintosh interface features including:

- *Windows* — events include activation, refresh, close, zoom in, zoom out, resize, and repositioned.
- *Buttons* — button clicks.
- *Cursors* — window, button and field determinations.
- *Edit and picture fields* — field clicks, clear, Return, Tab, and arrow key detection.
- *Keypresses* — key activation.
- *Operating system events* — includes suspend and resume application, clipboard changed, and mouse moved events.

- *Preview events* — special events sent by the runtime to your program before the dialog handler routines receives the actual event. Events include menu and field clicks, as well as growing, moving, zooming, and sizing windows,
- *Programmer-defined events* — special events sent by the program to itself.

We'll cover three of these dialog events in this chapter (buttons, windows, and cursors). The others will be introduced as they are required by *SimpleBase*.

As previously stated, events are messages from the operating system to the program. When the program receives an event, it identifies the event type and passes it on to an event handler. An **event handler** is a subroutine designed to handle a specific event type. *SimpleBase* has several event handler routines. Some are designed to deal with a specific event type (menu, dialog, cursor, etc.), yet others deal with a diverse cross-section of events (buttons, windows, cursors, etc.). With that in mind, let's examine the pseudocode in Program 47 that deals with dialog events.

PROGRAM 47. Main Loop event handling pseudocode.

```
HANDLEEVENTS receives and identifies event
HANDLEEVENTS sends event to correct Event Handler
Event Handler extracts information about event
Event Handler routes event to appropriate subroutine
Repeat until program ends
```

As you can see, the whole purpose of HANDLEEVENTS is to route an identified event to the correct event handler subroutine. It is the responsibility of the event handler to extract the information it requires to deal with the event. For purposes of this chapter, the event handler we'll talk about concerns dialog events.

Before we continue, we must point out that once an event is detected by HANDLEEVENTS and routed to an event handler subroutine, it must always be extracted from the event queue. Failure to extract an event can result in slow, unresponsive programs. Events which are not extracted create roadblocks in the event queue for new events. As new events arrive, they will eventually fill the event queue and push older events out of the queue, effectively losing them.

Regular Exercise

Okay, now that we understand dialog events and their usefulness in our programs, let's see how we deal with them in *SimpleBase*.

Specifying Event Handlers

In the Main Loop section of *SimpleBase* we specify the event handler where all dialog events should be sent. This notification tells HANDLEEVENTS where to route control when it detects dialog events in the event queue. We do this with an ON <event> FN statement like this:

```
ON DIALOG FN HandleDialogEvent
```

We add this line (as well as all other ON <event> FNs) just prior to entering the Main Loop. Our Main Loop begins to look something like this:

```
ON DIALOG FN HandleDialogEvent
ON MENU FN HandleMenuEvent
DO
  HANDLEEVENTS
UNTIL gQuit
END
```

Once the event handler for dialog events is specified, we add the dialog event handler subroutine as shown in Program 48.

PROGRAM 48. DIALOG event handling routine.

```
LOCAL FN HandleDialogEvents
  dlgEvnt% = DIALOG (0)
  dlgID% = DIALOG (dlgEvnt%)
END FN
```

Here we use the DIALOG function to return two values of importance. The first, dlgEvnt%, indicates the kind of dialog event retrieved from the event queue by HANDLEEVENTS. The second, dlgID%, is also returned by the DIALOG function, but its value is dependent upon the event type itself. For example, the _wndRefresh event returns the wndID% of the window that needs refreshing in dlgID%, and the _btnClick event returns a button ID in dlgID%. Each event carries additional information required by the program in its own dlgID%. We'll see how to handle several of these later in this chapter and others.

Window Handling

Since most dialog events involve windows or objects contained in windows (things like buttons and edit fields), it behooves us to respond to these events on a per window basis. This means that when we receive a DIALOG event, we determine which window the event was meant for, before deciding what to do with it.

While there are some common events that work with all windows (refresh being one), the events for buttons and edit fields are always specific to a particular window. In other words, an event meant for the Data Entry window should not interfere with an event for the find window.

By separating events by the window they apply to, we allow the program to deal with identical events differently. For example, a click in button #1 of the Data Entry window (**New Record**) will create a new record in the database. A click in button #1 of the find window (**Find**) will begin a search for matching text in the data file.

To discriminate among windows, modify the HandleDialogEvents function as shown in Program 49. As before, we pass off the handling of each window's dialog events to another subroutine, one that deals with that window's specific event requirements. We make use of the SELECT structure to call the program window the event occurred in. Note how we again make use of window constants to further improve readability.

See how we also use the class assigned to each window to identify which window event handler to call instead of the window's wndID%. For example: imagine that we had four identical windows open for data entry, each would have a different wndID%, but, each could have the same class type. By

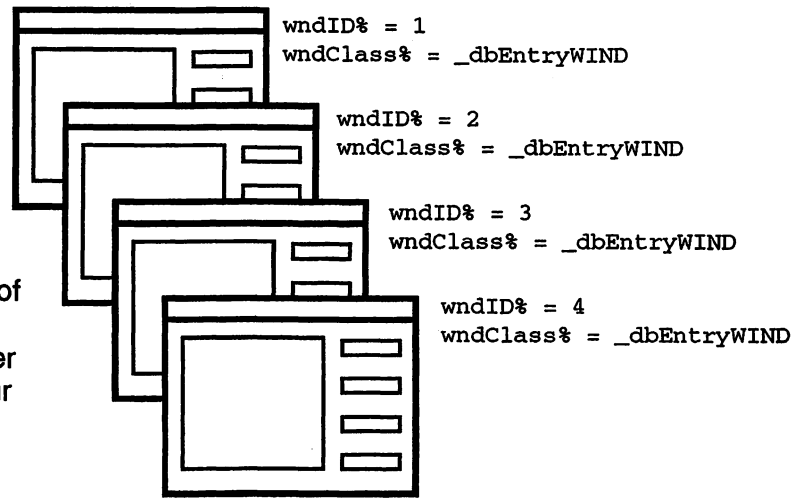
PROGRAM 49. DIALOG events for windows.

```
LOCAL FN HandleDialogEvents
  dlgEvnt% = DIALOG (0)
  dlgID% = DIALOG (dlgEvnt%)
  SELECT WINDOW (_outputWClass)
    CASE _dbEntryWIND : FN DialogEntryWindow (dlgEvnt%, dlgID%)
    CASE _dbFindWIND : FN DialogFindWindow (dlgEvnt%, dlgID%)
    CASE _aboutWIND : FN DialogAboutWindow (dlgEvnt%, dlgID%)
    CASE _helpWIND : FN DialogHelpWindow (dlgEvnt%, dlgID%)
    CASE _printWIND : FN DialogPrintWindow (dlgEvnt%, dlgID%)
    CASE _gotoWIND : FN DialogGotoWindow (dlgEvnt%, dlgID%)
  END SELECT
END FN
```

FIGURE 31. Window classes vs. window IDs.

Here are four windows with different ID's, however, they all have identical classes.

By trapping the wndClass instead of the `wndID%`, the same event handler can work for all four windows.



trapping the class, the same dialog subroutines can handle all dialog events in all four windows. See the diagram in Figure 31 for an example of using class types.

Each window's dialog handling routine requires both event values, so we pass along the `dlgEvnt%` and `dlgID%`. This enables them to respond accordingly no matter which event is passed to them.

Of course, the next step is to define the individual subroutines that deal with the window's events. Each accepts the two event parameters, and each uses another `SELECT` structure to deal with the `DIALOG` events passed. Program 50 shows the layout for the Data Entry window's dialog handling. Duplicate this example for each window in *Simplebase*.

Then we need to add some real capabilities to each dialog handling routine. Since our windows were created first, we'll start with window events. Our most important windows are defined, let's add some dialog event handling capabilities to deal with them.

PROGRAM 50. DIALOG window routine.

```
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW HANDLING EVENTS
    ' ••• BUTTON HANDLING EVENTS
    ' ••• CURSOR HANDLING EVENTS
    CASE ELSE
  END SELECT
END FN
```

Handling Window Events

The first category of dialog events deal with the windows themselves. There are numerous window related events, some of which help you determine if a window's close or zoom box were clicked, whether it needs to refresh itself, and others.

Within each window's event handler, we add `CASE` selections to deal with many of these events. Let's look at some of these subroutines so we can see exactly what is happening in each.

Window Close

When a program receives a `_wndClose` event, it first calls `FN WindowClose` which in turn calls `FN WindowCapture` to save any information entered by the user (text or button values), then closes the window for us. The same sequence is used for all of our windows whether they have data to save or not.

Active Window

The next event is `_wndClick`. When a `_wndClick` event is detected the program uses the `WINDOW` statement to make the chosen window frontmost on the screen. As the frontmost window, it now becomes the one ready to accept all text or graphic commands.

Subsequently, whenever a window is brought to the front, a series of activation and refresh events are generated. These additional events are very useful as we will see in the next couple of sections.

Window Activation

The `_wndActivate` event is generated whenever a window is brought to the front or sent behind another window. This event makes our program more interactive. We can use this event to update information displayed in the window, save a window's contents for later, or update program menus.

In *SimpleBase*, we'll use the activation of a window to update our menus. Whenever a window receives a `_wndActivate` event, it calls `FN UpdateMenus` to keep our menus in sync. Since we're only interested in window activation events, `FN UpdateMenus` only checks for the absolute value of the `wndID%` and updates the menus according to which window is frontmost on the screen.

Window Refreshing

With a `_wndRefresh` event, the system tells the program that a portion of a window needs updating. It could be that the window has just been created and needs to be drawn for the first time, or that it was partially obscured by

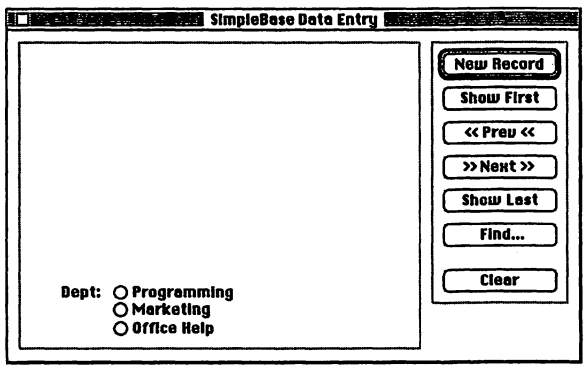
PROGRAM 51. Window activation updates menus.

```
LOCAL FN MenuUpdate (wndID%)
  SELECT ABS(wndID%)
    CASE _dbEntryWIND
      MENU _mFile, 0, _enable
      MENU _mRecord, 0, _enable
    CASE _dbFindWIND, _printWIND, _gotoWIND
      MENU _mFile, 0, _disable
      MENU _mRecord, 0, _disable
    CASE _aboutWIND, _helpWIND
      MENU _mFile, 0, _disable
      MENU _mEdit, 0, _disable
      MENU _mRecord, 0, _disable
    CASE ELSE
      MENU _mFile, 0, _enable
      MENU _mRecord, 0, _enable
  END SELECT
END FN
```

another window, or even hidden entirely by another program (System 7 or MultiFinder). This is our cue to redraw all of a window's changing elements. Changing elements are those parts of a window not automatically updated by the runtime itself. What's included in this? All buttons, edit and picture fields, and scroll buttons are already taken care of by the runtime. Your responsibility is to refresh any text or graphics placed there by the program. These are things like data, borders, icons, lists, etc.

For example, in the Data Entry window, I wanted a nice little border around both the data entries and the buttons Therefore, I added a bit of code in the `_wndRefresh` section to draw some borders.

FIGURE 32. The Data Entry window after a refresh.



See Figure 32 for a look at the Data Entry window's new borders. The code to create the borders in FN DialogEntryWindow is shown in Program 52 as well as the rest of the window event code. Just to see if window refreshing works, add this section of code to *SimpleBase* and try it out. Start by opening the Data Entry window. Briefly open the Find window and then close it. Portions of the borders may disappear, but as soon as the _wndRefresh event is received, FN DialogEntryWindow redraws them good as new.

PROGRAM 52. Data window refreshing.

```
LOCAL
DIM rect;8
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndRefresh
      PEN ,,,,3
      CALL SETRECT (rect, 10, 10, 360, 280)
      DEF TITLERECT ("", 0, rect)
      CALL SETRECT (rect, 370, 10, 490, 240)
      DEF TITLERECT ("", 0, rect)
      PEN ,,,,0
    CASE ELSE
  END SELECT
END FN
```

Handling Button Events

Dealing with button events is pretty straightforward. The runtime interprets the system event and gives you the event type _btnClick, and the button ID. A window's dialog handler that contains any buttons must trap for the _btnClick event.

Button Handling

Program 53 shows how the button handling is done in the Data Entry and Find windows (the other sections have been removed to save space). Again, we make use of the trusty SELECT structure to deal with the myriad of events a window must handle. Since we've already created subroutines to deal with the actions in the button titles, it's a simple matter to connect the buttons to the subroutines.

Back in the "Menus" chapter we introduced two methods of accessing a program subroutine. The user chooses a menu item with the mouse or uses a command key equivalent from the keyboard. Now, with the introduction of

PROGRAM 53. Window button handlers.

```
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• BUTTON HANDLING SECTION
    CASE _btnClick
      SELECT dlgID%
        CASE _newRecBTN    : FN ItemNew
        CASE _firstRecBTN  : FN ItemFirstRecord
        CASE _prevRecBTN   : FN ItemPrevRecord
        CASE _nextRecBTN   : FN ItemNextRecord
        CASE _lastRecBTN   : FN ItemLastRecord
        CASE _clearRecBTN  : FN ItemClearRecord
        CASE ELSE
          FN RadioButtonHandler (_programBTN, _officeBTN, dlgID%)
      END SELECT
    CASE ELSE
  END SELECT
END FN
```

dialog event handling in a window, a third method presents itself, the user can click on a button to call the same subroutine.

Data Entry Window Events

As you can see, most of the routines that deal with the Data Entry window's push buttons match already defined menu routines. This re-use of code makes for a compact and efficient program design. Also, writing one piece of code that is called from many points in the program makes the interface more flexible.

This benefits the user since a click on a button, a menu choice by mouse, or a command key, accesses the same subroutine.

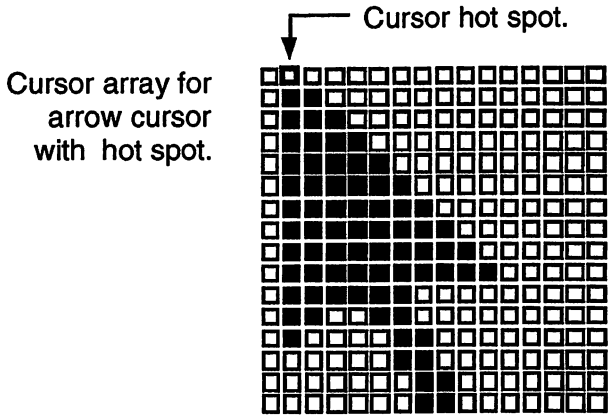
The final `CASE ELSE` in the Data Entry's button handling section calls the `RadioBtnHandler` subroutine to deal with the radio buttons in the window. In the Find window, the `CASE ELSE` calls the `CheckboxHandler` subroutine.

Notice how easy it is to read the statements in this subroutine. Using the constants and descriptive function names makes the code very readable, even without any descriptive comments. We'll see how to add the rest of these routines dialog handling routines later on in the chapter.

Handling Cursor Events

The last type of dialog event we will deal with is cursor events. We start with a little bit of background and then show the code.

FIGURE 33. Cursor array.



What is a Cursor?

A **cursor** is any 256-byte image bounded by a 16-by-16 bit square. The mouse driver normally displays the current cursor and handles the details of moving it on the screen. A cursor can be black and white or color and is typically stored as a resource in an application.

The cursor is always moved on the screen relative to the path that the mouse is moved by the user. A cursor's **hot spot** is the point of the cursor which is reported to the operating system when a user clicks the mouse button.

Your application is responsible for changing the cursor as the user moves it. For example, applications normally change the cursor to an I-beam whenever the mouse location intersects an active edit field. Others change the cursor when it's over a button. Still others use mouse down or up events to change the cursor's appearance. Which ones you'll use will depend on your program's requirements. We will examine two FB cursor events: `_cursEvent` and `_cursOver`.

Dealing with Cursors

The `_CursEvent` event tells the application when a cursor has entered or exited a window. Since we have no need for a special cursor in our windows, we just ensure that it becomes the default arrow cursor when it's in one of our windows. We do that by placing the following line in each window's dialog handling subroutine:

```
' ••• CURSOR HANDLING SECTION
CASE _cursEvent : CURSOR _arrowCursor
```

PROGRAM 54. Cursor handlers.

```
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' CURSOR EVENT HANDLERS
    CASE _cursEvent
      CURSOR _arrowCursor
    CASE _cursOver
      SELECT dlgID%
        CASE < 0 ' handle edit fields here
          LONG IF ABS(dlgID%) = WINDOW (_efNum)
            CURSOR _iBeamCursor
          XELSE
            CURSOR _arrowCursor
          END IF
        CASE > 0 ' handle buttons here
          CURSOR _crossCursor
        CASE ELSE ' not over button or edit field
          CURSOR _arrowCursor
      END SELECT
    CASE ELSE
  END SELECT
END FN
```

Once we know the cursor is in a program window, we can detect when it's over a button or edit field using `_cursOver`. The code to handle a cursor over both objects can be seen in Program 54.

Note that we must distinguish between buttons and edit fields by checking the value of the `dlgID%`. A positive value indicates the cursor is over a button. For now, we'll just change the cursor shape to a plus. Later, Peak Performance we'll use *ResEdit* to add a custom cursor to this bit of code.

A negative value in `dlgID%` indicates an edit or picture field. When the cursor is over an edit or picture field, the program should change the cursor to an I-beam shape. However, it should only do this for the active edit field. All others should be ignored until they are made active. To determine the active field, we use the `WINDOW (_efNum)` function to return the ID of the currently active field. If the two match, we change the cursor to an I-beam. We'll see how to handle this in the chapter "Edit & Picture Fields".

Because so many of the cursor events are handled identically, no matter which window is open, it seems appropriate to create a single subroutine to handle these events.

Well, we've covered all the event routines required for the data entry window up to this point. Using the same technique you can create routines to handle the remaining windows in *SimpleBase*. See the complete program in the back of the book for complete details. Once you put in all the routines, run *SimpleBase* and test out the various additions we've made to the source.

Window Dialog Handlers

Now that we have all of our window, button, and cursor dialog event handling routines in place, let's see how each window's dialog handler is setup.

Data Window Handler

The Data Entry window code is shown in Program 55 has the most event handling features. Besides the normal window events like activation and closing, it also requires some `_wndRefresh` handling. It does a multitude of button events to tend to as well as cursor events for all the fields. While no more difficult then all the other windows, it does have more features to contend with.

Find Window Handler

In Program 56 we show how to handle the dialog events for the Find window. Here we have the standard window dialog events (`_wndClose`, `_wndActivate`, etc.), a checkbox for the search strings case setting, and a cancel push button and find shadow button.

PrintWindow Handler

The Print window is unremarkable as shown in Program 57. The dialog handling code contains a shadow and push button, and three radio buttons to select how records are printed. Later, when we add some edit fields it'll become more exciting.

Goto Window Handler

Goto is as easy as it gets as demonstrated in Program 58, with dialog handling for two buttons: a push button to cancel and a shadow button to implement the goto record action.

Help Window Handler

Finally, the Help window code is shown in Program 59. It has just two plain push buttons to cycle through the help text we will soon add. Unlike most of the other windows, we get rid of this window only with the window's close box.

PROGRAM 55. Entry window handler.

```
LOCAL
DIM rect;8
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose : FN WindowClose (_dbEntryWIND)
    CASE _wndActivate : FN UpdateMenus
    CASE _wndClick : WINDOW #_dbEntryWIND
    CASE _wndRefresh
      PEN ,,,,3
      CALL SETRECT (rect, 10, 10, 360, 280)
      DEF TITLERECT ("", 0, rect)
      CALL SETRECT (rect, 370, 10, 490, 240)
      DEF TITLERECT ("", 0, rect)
      PEN ,,,,0
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _newRecBTN : FN ItemNew
        CASE _firstRecBTN : FN ItemFirstRecord
        CASE _prevRecBTN : FN ItemPrevRecord
        CASE _nextRecBTN : FN ItemNextRecord
        CASE _lastRecBTN : FN ItemLastRecord
        CASE _findRecBTN : FN WindowBuild (_dbFindWIND)
        CASE _clearRecBTN : FN ItemClearRecord
        CASE ELSE
          FN RadioBtnHandler% (_programBTN, _officeBTN, dlgID%)
      END SELECT
    ' ••• FIELD EVENTS
    ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN
```

PROGRAM 56. Find window handler.

```
LOCAL FN DialogFindWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose : FN WindowClose (_dbFindWIND)
    CASE _wndActivate : FN UpdateMenus
    CASE _wndClick : WINDOW #_dbFindWIND
    CASE _wndRefresh
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _ignoreCaseBTN : FN CheckBoxHandler% (dlgID%)
        CASE ELSE : FN WindowClose (_dbFindWIND)
      END SELECT
    ' ••• FIELD EVENTS
    ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN
```

PROGRAM 57. Print window handler.

```
LOCAL FN DialogPrintWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose : FN WindowClose (_printWIND)
    CASE _wndActivate : FN UpdateMenus
    CASE _wndClick : WINDOW #_printWIND
    CASE _wndRefresh
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _thisRecBTN,_allRecBTN,_selectRecBTN
        CASE ELSE : FN WindowClose (_printWIND)
      END SELECT
    ' ••• FIELD EVENTS
    ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent : FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN
```

PROGRAM 58. Goto window handler.

```
LOCAL FN DialogGotoWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose : FN WindowClose (_gotoWIND)
    CASE _wndActivate : FN UpdateMenus
    CASE _wndClick : WINDOW #_gotoWIND
    CASE _wndRefresh
    ' ••• BUTTON EVENTS
    CASE _btnClick : FN WindowClose (_gotoWIND)
    ' ••• FIELD EVENTS
    ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN
```

PROGRAM 59. Help window handler.

```
LOCAL FN DialogHelpWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose : FN WindowClose (_helpWIND)
    CASE _wndActivate : FN UpdateMenus
    CASE _wndClick : WINDOW #_helpWIND
    CASE _wndRefresh
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _prevHelpBTN
        CASE _nextHelpBTN
      END SELECT
    ' ••• FIELD EVENTS
    ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
     FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN
```

We'll have some minor refresh handling to take care of as well as handling a scroll button and resizeable edit field. More on that later in the book.

Peak Performance

This section will explain how to create custom cursors for use in *SimpleBase* and other programs.

Cursor Designing

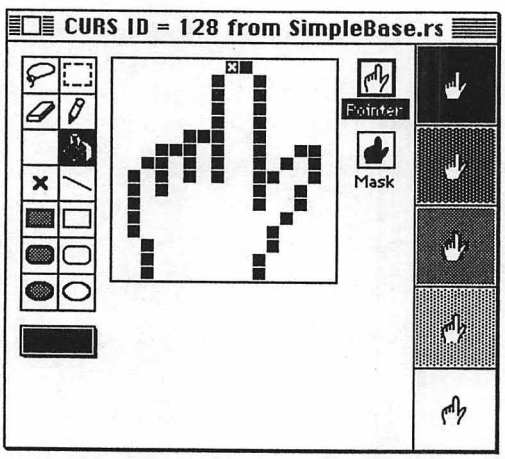
We start our custom cursor design by opening *SimpleBase.rsrc*. Once open, choose **Create New Resource** from the **Resource** menu. At the **Select New Type** dialog, enter or choose from the scrolling list the CURS type. Click **OK**. *ResEdit* creates a new CURS resource and opens the cursor editor window as shown in Figure 34.

Designing Cursors

As soon as the cursor editor is open, use it to design both the cursor and the cursor mask. Remember, the cursor mask is what allows you to view the cursor against different screen backgrounds, so don't forget to add one.

The palette on the left of the window contains all of the graphic tools needed to create custom cursors. Use the [×] tool to place the cursor's hot spot in the design. There are two menus: **Transform** and **CURS**, that enable you to manipulate the cursor in various ways (Flip Horizontal, Flip Vertical, etc.) and try the new cursor design. Once you are satisfied with your cursor design, save your work.

FIGURE 34. Design CURS with cursor editor.



Using Custom Cursors

Next, we modify the program code to deal with the new cursor. In this case, a new constant is called for in the globals file. Open *SimpleBase.glbl* and add the following constant:

```
' >>> OTHER CONSTANTS
_fingerCursor = 128
```

Save the changes and close the globals file. Next, open *SimpleBase*. Locate the dialog routines for all of the windows that contain buttons. Find the `_cursOver` section of the event handler and change:

```
LONG IF ABS(dlgID%) = WINDOW (_efNum)
  CURSOR _IBeamCursor
XELSE
  CURSOR _plusCursor
END IF
```

to:

```
LONG IF ABS(dlgID%) = WINDOW (_efNum)
  CURSOR _IBeamCursor
XELSE
  CURSOR _fingerCursor
END IF
```

Save your changes and try out the program. In place of the plus cursor, you should see the finger cursor appear whenever the mouse cursor is over an active button. Now that you know the procedure, you can create your own custom cursor library.

Event Handling

There are two means of retrieving events from the event queue. Under System 6 (and not running MultiFinder), events are retrieved using `GetNextEvent`. Since only one application can operate at a time under System 6, the application naturally collects all events issued by the operating system.

However, under System 6 with MultiFinder and System 7, multiple applications can be running at once. Normally, the foremost application (the one you are working on) collects the majority of event messages. Applications operating in the background can also receive events and perform tasks. This is what enables system utilities like Print Monitor to print in the background while you continue to add more chapters to your book.

In order to share events, programs must use a different mechanism to retrieve events. Instead of `GetNextEvent`, it must use `WaitNextEvent`. We can tell the runtime to use `WaitNextEvent` by setting the number of times per second the program requires events.

The code in Program 60 detects which system the program is operating under and sets the tick count to what our program requires. Note that the code only checks for the presence of System 7, as there is no Apple approved method of detecting the presence of MultiFinder.

PROGRAM 60. Enabling background processing.

```
ticksPerSecond = 6
LONG IF SYSTEM (_sysVers) > 699
  & EVENT - 8, ticksPerSecond
END IF
```

Suspend and Resume Events

Both System 7 and System 6's MultiFinder allow multiple applications to run concurrently. The user can switch between running applications by clicking on any portion of a visible window from another application, or choose the application directly from the Application's menu (System 7) or the menu (System 6).

With the capability of having multiple applications open at the same time, it's to your benefit to write programs that can operate in this environment. One way to do that is to detect the `_mfSuspend` and `_mfResume` events.

When a program is about to be switched behind another, the operating system sends it a `_mfSuspend` event. This is a signal that it must prepare itself to be switched out as the foreground application.

The foreground application is the one currently in use by the user. All other running applications are called background applications. A program should take this signal as an opportunity to convert any private clipboard contents to the system clipboard.

The FB runtime handles the conversion of the private text scrap for you, but only if it contains simple ASCII text. If your program manipulates other types of data, like pictures for example, you must place that data on the clipboard yourself.

Additionally, some programs take the opportunity to hide all palette-style windows before the switch. You can do this with your program windows by using a routine similar to the one shown in Program 61. The routine shows how to hide and show program windows. You'll need to modify it to remember only the windows that were open when the program was switched out. Otherwise, all of the program windows will appear when this routine is called on to show windows.

PROGRAM 61. Hide and Show Windows.

```
LOCAL FN HideShowWindows (hideFlag%)
  FOR wndID% = 1 TO 63
    LONG IF WINDOW (-wndID%)
      LONG IF hideFlag% = _mfSuspend
        WINDOW #-wndID%
      XELSE
        WINDOW #wndID%
      END IF
    END IF
  NEXT wndID%
END FN
```

The opposite happens when a program receives a `_mfResume` event. This tells the program that it is about to be moved in front of all other applications. When a `_mfResume` event is received, the program should accept this as a signal to convert the system clipboard to its own internal scrap. Again, if the program can import types of data other than text, you must handle the importing process yourself.

Cooldown

In this chapter we learned more about events and how to respond to three different types of dialog events for windows, buttons, and cursors. Along the way we saw how to react to dialog events on a window-by-window basis.

We learned how to handle the common window events, output, activate, refresh, and close. We also saw how to deal with push buttons, and implemented our checkbox and radio button handler routines. Finally, we not only saw how to detect and respond to cursor events, but we saw how to create a custom cursor for our program.

CHAPTER 8

Edit & Picture Fields

Warm-up

This chapter introduces you to edit and picture fields. In this chapter you will learn:

- What edit and picture fields are,
- To identify the edit field types,
- Two methods of inserting data into edit fields,
- How to extract text data from edit fields,
- How to insert images into a picture field,
- How to close both edit and picture fields, and
- How to deal with different field events.

What are Edit Fields?

A **edit field** is an area within a window that displays static or editable text. Text fields are normally editable, but can be static, disabled, or even inverted. Edit fields can display text in any available font, and in any style, size, or color. When linked to a scroll button, the contents of an edit field are scrolled when the user clicks on the scroll button. You can have up to 8192 fields, either edit or picture versions within a single window, memory permitting.

All edit fields that allow text entry have an insertion point. An **insertion point** is the position within the edit field where the next character will be inserted when a key is pressed. It is usually indicated by a blinking caret in

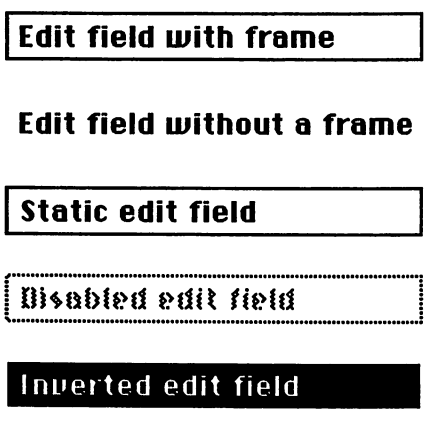
the form of a vertical bar (|). Static, non-editable edit fields do not display a caret.

To act upon the text within an edit field, a selection must be made. A **selection** is a sequence of one or more characters chosen by the user or the program for editing. A selection is indicated by **highlighting** the current selection, either by inverting or coloring the selection. Once text has been selected, the user can cut, copy, paste, or clear the selected text, or change the font characteristics to any available font, size, style, or color.

Positioning the insertion point can be done using the mouse, the arrow keys, or under program control. When the user clicks the mouse in an active edit field, the insertion point is positioned at the nearest character. The user can then use the arrow keys to move the insertion point. The left and right arrow keys move the insertion point one character forward or back. The up and down arrow keys move the insertion point one line up or down, respectively.

If the cursor is positioned at the start or end of the field's text, an arrow key event is generated that can be detected using the `DIALOG` function.

FIGURE 35. Standard edit field types.



Edit Field Features

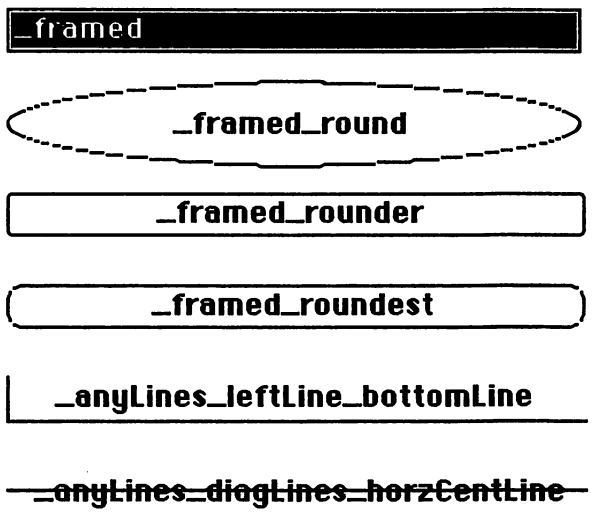
Edit fields fall into two broad categories: those that accept carriage returns and those that don't. When a field that accepts carriage returns is active, any press of the Return key sends a carriage return to the field. Any active field that doesn't accept carriage returns generates an `_efReturn` event instead.

Edit fields also come in five standard types: framed, noFramed, static, gray (for disabled), and inverted. By mixing these various types you can create edit fields that can serve any purpose a program might require. For example, normal single line edit fields used for data input do not accept carriage

returns and have frames. Edit fields used to display titles are usually static and not framed. Some examples of the various standard field types can be seen in Figure 35.

You can further customize an edit field's appearance by adding one of five modifiers to the field type. These modifiers are: `_round`, `_rounder`, `_roundest`, `_boldBox`, and `_anyLines`. Examples of these types are shown in Figure 36.

FIGURE 36. Edit field type modifier examples.



Here is an example of using the custom type options to modify the appearance of an edit field.

Note the `_anyLines` option in the bottom two examples. This allows you to customize an edit field frame.

A word of caution though: use these custom field types only when absolutely necessary to prevent causing confusion in your users. Users expect edit fields to look like edit fields, not examples of cleverness. If you do use them, you will need to experiment with the various options to get the effect you want. Make sure that a user can immediately identify them as places to enter text.

- *Note, it may take some experimenting to get the custom field type that you want, as some modifiers interfere with other modifiers.*

Unstyled vs. Styled

There are two kinds of edit fields: unstyled and styled. The original unstyled edit field dates from the early days of the Macintosh and allows the field to contain exactly one font, size, and style for the entire field. You create an unstyled edit field by using a positive `fieldID%` like this:

```
EDIT FIELD #fieldID%, tmp$, rect, type, just
```

With the introduction of the color Macintosh, a new styled edit field became available. You can create a styled edit field by using a negative `fieldID%` like this:

```
EDIT FIELD #-fieldID%, tmp$, rect, type, just
```

The runtime recognizes the difference and builds the correct type of field. You can see the differences between the two field kinds in Figure 37. Note that both unstyled and styled fields are always referred to using positive `fieldID%` numbers no matter how they were defined. Any attempt to refer to an edit field by a negative ID will cause a runtime error.

FIGURE 37. Styled vs. unstyled edit fields.

Create unstyled edit fields with a positive `fieldID%`:

A standard unstyled edit field can only have one font, size, style, and color associated with it.

Create styled edit fields with a negative `fieldID%`:

A styled edit field can have multiple **fonts**, sizes, *styles* , and color associated with it.

You define the initial text font, size, style, and color when you use the `TEXT` statement to specify a window's default font and font attributes. Thereafter, you can change a field's font and associated attributes using the `EDIT TEXT` statement. For example, to change the font for selected characters in the active edit field from **Chicago** to Geneva do this:

```
SETSELECT startChar%, endChar%
EDIT TEXT _geneva
```

Creating Edit Fields

Once a program window has been built, putting in edit fields is not difficult. We start by assigning the field a `fieldID%` number (either positive or negative for unstyled or styled fields respectively), define its location in the window, assign it a field type and justification, and optionally add a class specifier.

The second parameter can be a string variable, a quoted string, a text handle, or a `TEXT` resource ID. Usually, for text of less than 255 characters in length, you'll use either string variables (`tmp$`) or quoted strings (`"this is a string"`). For longer text, you must use text handles or resource IDs. See the Peak Performance section for details on dealing with large blocks of text.

Next, define the location of the edit field in window coordinates using the `(left, top)-(right, bottom)` format. Then, define the field type using one

PROGRAM 62. Creating Edit Fields.

```
WINDOW #1, "EDIT FIELD", (0,0)-(500,300), _doc
TEXT _sysFont, 12
tmp$ = "An example of an edit field"
EDIT FIELD #-1, tmp$, (10,10)-(300,300), _framed, _leftJust
DO
UNTIL INKEY$ <> ""
```

of FB's constants, and finally assign a justification to the field. Examine Program 62 to see how to define a single edit field in a window.

Setting Field Data

While many edit and picture fields will have their data set once when the field is created and never be changed for the life of the window, others will require updates to reflect changing conditions. This is the case with the fields in the Data Entry window used to display record information. As the user moves forward or backward through the data file, the fields will display the current viewable record. Update the field data using either `EDIT FIELD` or `EDIT$`.

Use `EDIT FIELD` when the replacement text should be selected or highlighted after the insertion. Use `EDIT$` when you want to replace the text in an edit field without it being selected. To demonstrate:

```
WINDOW #1 : TEXT _sysFont, 12
EDIT FIELD #-1, "", (10,10)-(300,25), _framed
EDIT FIELD #-2, "", (10,40)-(300,55), _framed
tmp$ = :This is an example of text insertion."
EDIT$(1) = tmp$
EDIT FIELD #2, tmp$
```

This example will insert the same text into both fields, but the text inserted into field #1 will not be selected and that inserted into field #2 will be selected. Examine Figure 38 to see the difference.

There are a couple of variations to inserting text into an edit field that are useful in certain circumstances. First, both `EDIT FIELD` and `EDIT$` can accept handles and resource IDs to `TEXT` resources. You can load a text handle into an edit field using either of these two methods:

```
EDIT FIELD #fieldID%, &zTxtH&
EDIT$ (fieldID%) = &zTxtH&
```

The only caveat is that the handle must be properly formatted in order for this insertion to work. This means that the first two bytes of the handle must contain an integer value representing the number of characters in the handle.

See "Using all 32K" later in the Peak Performance section for details on this format.

Use the same two methods to access TEXT resources:

```
EDIT FIELD #fieldID%, %textResID%
EDIT$ (fieldID%) = %textResID%
```

Find out more about this text insertion method in the chapter "Strings & Text".

Formatting Text

Besides text, you may want to insert numerical data into a field. Since a field requires text, use STR$ or USING to convert the numerical value to a string representation of the number. For example: to convert a numerical value into its string, use the STR$ function like this:

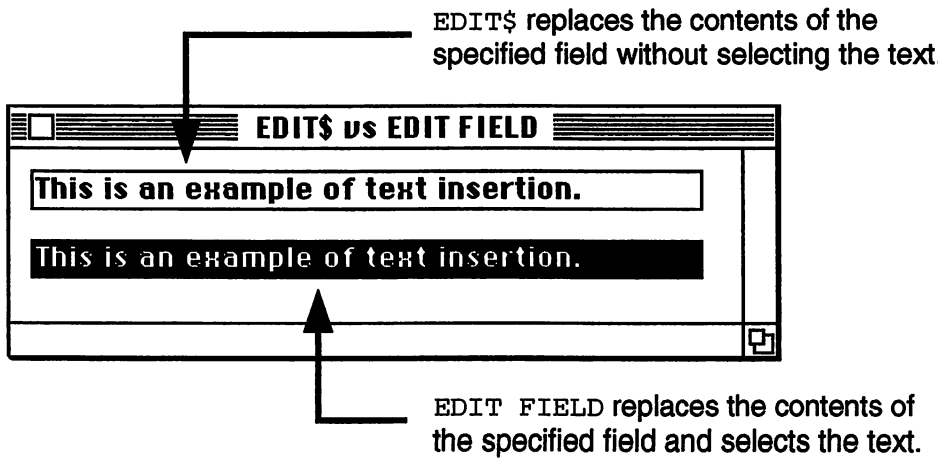
```
userData$ = STR$(123.456)
EDIT$ (15) = userData$
```

You can format numerical data before displaying it with the USING function like this:

```
userValue! = 123.456
EDIT$ (15) = USING "###.###", userValue!
```

While EDIT$ will work for both edit and picture fields to replace a field's content, you must use PICTURE FIELD to replace a picture field's image that should be selected. Additionally, you can load text resources into edit fields using a resource ID or a text handle with both EDIT$ and EDIT FIELD.

FIGURE 38. Inserting field data.



Getting Field Data

Once the user has entered data into an edit field, it's often necessary to extract that information. You can retrieve an edit field's data using the `EDIT$` function. For example, to extract the text from an edit field whose `fieldID%` is 15, do this:

```
userData$ = EDIT$(15)
```

Assuming that a field's data isn't just text, but also has numbers, you can get the value of the field data like this:

```
userValue% = VAL(EDIT$(15))
```

Note that these two methods only work when the text in the field is under 255 characters in length. Since FB internally uses Pascal formatted strings, the standard string variable can't handle anything larger.

To handle larger blocks of text, you must delve into the field's record structure and work directly with the text data itself. To see how to do that, refer to "Using all 32K" later in this chapter.

Closing Edit Fields

You can close edit fields using the `EDIT FIELD CLOSE` statement with the appropriate `fieldID%`. Remember that the text information stored in the field will be gone forever once the field is closed. To close an edit field do this:

```
EDIT FIELD CLOSE #fieldID%
```

Normally, you can let the FB runtime handle all the details of closing both edit and picture fields, as well as buttons, when it closes a window. This is just another item taken care of by the FB runtime.

Enabling & Disabling Fields

You may occasionally have to disable fields in a window. You can do this by resetting the field type. For example, if we have a `_framedNoCR` field, we can disable it like this:

```
EDIT FIELD #fieldID%, , , _statFramedGray
```

And reactivate it using the original field type like this:

```
EDIT FIELD #fieldID%, , , _framedNoCR
```

What are Picture Fields?

A **picture field** is an area within a window that displays picture images. Picture fields are usually static, but can be active and act as graphic buttons *a la* HyperCard™. Picture fields share the same data structure in memory

that allows up to 8192 edit, picture, or any combination of these two field types in a single window. A picture field cannot have the same `fieldID%` as an edit field in the same window.

Loading Pictures

Picture fields share the same definition format used by edit fields. The only real difference is that one operates on text, the other on picture data.

A picture field can display images by handle, resource ID, or resource name. For example, to create a simple picture handle and display it on screen examine the program shown in Program 63.

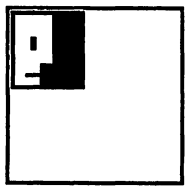
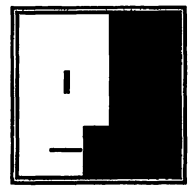
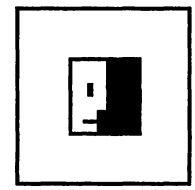
PROGRAM 63. Picture field demonstration.

```
WINDOW 1, "PICTURE FIELD DEMO"
' ••• CREATE A PICTURE
PICTURE ON
COLOR _zRed
CIRCLE FILL 100, 100, 50
CIRCLE 100, 100, 55
COLOR _zBlack
PICTURE OFF, pictH&
' ••• NOW DISPLAY IT IN A PICTURE FIELD
PICTURE FIELD #1, &pictH&, (10,10)-(200, 200), _framed, _cropPICT
DO
   HANDLEEVENTS
UNTIL 0
END
```

Note the program makes use of the `PICTURE ON` and `PICTURE OFF` statements to create a picture handle. We then pass the handle to the `PICTURE FIELD` and let it do the rest of the work.

A picture field's justification parameter defines how the graphic will display inside the field. The three types of graphic justifications are shown in Table 7.

TABLE 7. PICTURE FIELD justifications.

_cropPict	_scalePict	_centerPict
		

The _centerPict justification requires that the top-left corner of the picture frame (not the bounds of the PICTUREFIELD) always be set to 0, 0. Anything else causes the field to scale the image and center it on the lower-right corner of the picture field frame.

Creating Picture Fields

Picture fields use syntax identical to edit fields for their creation. You assign the picture field a fieldID% (different than any edit fields in the same window), pass it a picture identifier (handle, name, or resource ID), specify an area in the window, then assign a field type, and justification.

Setting Picture Fields

Just as was done with edit fields, use the PICTURE FIELD and EDIT$ statements to assign images to a picture field. The only difference is that you must tell the runtime how to interpret the image data. For example, to assign a picture handle to a picture field do this:

```
PICTURE FIELD #fieldID%, &pictH&
```

or this:

```
EDIT$(fieldID%) = &pictH&
```

The ampersand (&) before the pictH& tells the runtime that the number following represents a picture handle, while a leading percent sign (%) uses a use the PICT resource ID to places it into the picture field:

```
PICTURE FIELD #fieldID%, %pictResID%
```

or like this:

```
EDIT$(fieldID%) = %pictResID%
```

Finally, place a resource PICT using the resource's name like this:

```
PICTURE FIELD #fieldID%, pictResName$
EDIT$(fieldID%) = pictResName$
```

Closing Picture Fields

Just as you can with edit fields, you can close picture fields using the EDIT FIELD CLOSE statement. All you need is the appropriate fieldID and a line of code like this:

```
EDIT FIELD CLOSE #pictFieldID%
```

Closing a window also closes all of the field and button structures used in the window. Remember to always extract your data first.

PROGRAM 64. Field constant definitions.

```
' >>> DATA ENTRY WINDOW
_dbNameFLD      = 1
_dbAddrFLD      = 2
_dbCityFLD      = 3
_dbStateFLD     = 4
_dbZipFLD       = 5
_dbPhoneFLD     = 6
_dbFaxFLD       = 7
_dbPhotoFLD     = 11
' >>> FIND WINDOW
_searchFLD      = 1
' >>> HELP WINDOW
_helpFLD        = 1
' >>> PRINT WINDOW
_firstFLD        = 1
_lastFLD         = 2
' >>> GOTO WINDOW
_gotoFLD        = 1
```

Regular Exercise

Now that we understand edit and picture fields much better, it's time to add them to our program. Of course we begin by defining the field constants that will make life easier for us. The field constants used by *SimpleBase* are shown in Program 64. You'll note that there are only constants for active and informational display fields, fields used simply to display titles or instructions are left undefined.

Creating EDIT FIELDs

With the constants in place, it's time to add the fields to our various windows. The natural place to start is each window's build routine. There, after all button definitions, we begin adding both the edit and picture fields.

Data Entry Window

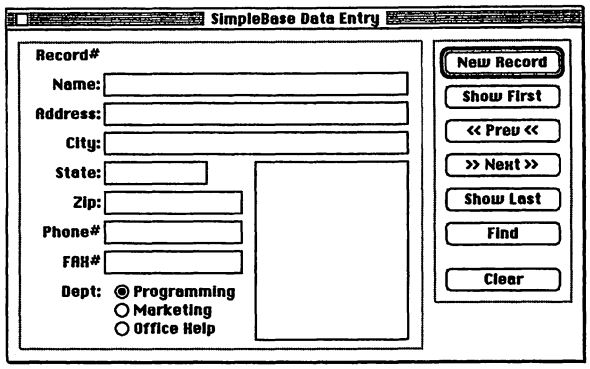
The Data Entry window has the majority of edit fields and picture fields, so we'll start there. We begin by creating the static edit fields that display the field name, add the picture field, followed by the editable fields, and finish by setting the first editable field active. The lines to add to `FN BuildEntryWindow` are shown in Program 65.

PROGRAM 65. Adding fields to the Data Entry window.

```
' ••• STATIC TEXT FIELDS
xPos% = 85
tmp$ = "Name:"
EDIT FIELD #21, tmp$, (20,40)-(xPos%xPos%-5,56), _statNoframed, _rightJust
tmp$ = "Address:"
EDIT FIELD #22, tmp$, (20,66)-(xPos%-5,82), _statNoframed, _rightJust
tmp$ = "City:"
EDIT FIELD #23, tmp$, (20,92)-(xPos%-5,108), _statNoframed, _rightJust
tmp$ = "State:"
EDIT FIELD #24, tmp$, (20,118)-(xPos%-5,134), _statNoframed, _rightJust
tmp$ = "Zip:"
EDIT FIELD #25, tmp$, (20,144)-(xPos%-5,160), _statNoframed, _rightJust
tmp$ = "Phone#"
EDIT FIELD #26, tmp$, (20,170)-(xPos%-5,186), _statNoframed, _rightJust
tmp$ = "FAX#"
EDIT FIELD #27, tmp$, (20,196)-(xPos%-5,212), _statNoframed, _rightJust
tmp$ = "Dept:"
EDIT FIELD #28, tmp$, (20,222)-(xPos%-5,238), _statNoframed, _rightJust
tmp$ = "Record#"
EDIT FIELD #29, tmp$, (20,14)-(xPos%-5,30), _statNoframed, _rightJust
EDIT FIELD #30, ""  , (xPos%,14)-(330,30), _statNoframed, _leftJust
' ••• STATIC PICT FIELDS
PICTURE FIELD #_dbPhotoFLD, "", (215,118)-(345,270), _statframed, _cropPict'
••• ACTIVE EDIT FIELDS
EDIT FIELD #-_dbNameFLD, "", (xPos%,40)-(345,56),_framedNoCR, _leftJust
EDIT FIELD #-_dbAddrFLD, "", (xPos%,66)-(345,82),_framedNoCR, _leftJust
EDIT FIELD #-_dbCityFLD, "", (xPos%,92)-(345,108),_framedNoCR, _leftJust
EDIT FIELD #-_dbStateFLD, "", (xPos%,118)-(170,134),_framedNoCR, _leftJust
EDIT FIELD #-_dbZipFLD , "", (xPos%,144)-(200,160),_framedNoCR, _leftJust
EDIT FIELD #-_dbPhoneFLD, "", (xPos%,170)-(200,186),_framedNoCR, _leftJust
EDIT FIELD #-_dbFaxFLD, "", (xPos%,196)-(200,212),_framedNoCR, _leftJust
EDIT FIELD #_dbNameFLD                          'set active edit field
```

Note the use of the `xPos%` variable. This is a little trick you can use to align the right edge of a title field with the left edge of its editable field. This is very handy when adding fields to a window. Since I often need to fine tune the location of the fields so that all the text in a title field fits, assign a variable that defines the boundary between the two fields. Then if you need to resize either one, just change one variable and the entire window is reformatted automatically. The completely designed Data Entry window can now be seen in Figure 39.

FIGURE 39. Final Data Entry window.



Other Windows

The remaining windows have their own requirements for editable and static fields. Most follow the same pattern as the Data Entry window. Add the static fields first, then follow with the editable fields. For a complete listing, be sure to examine the program in the back of the book.

Help and About Windows

You may have noticed the absence of the Help and About windows in the previous paragraphs. The Help window will be dealt with in the chapter "Scroll Buttons" and the chapter "Strings & Text", while the About window will be fully described in the chapter "Alerts".

Feel free to try out *SimpleBase* and see all your new edit and picture fields. Remember, continuous testing is the main method of discovering program bugs, and taking steps to solve them.

Handling Field Events

All right, now that our fields are in place, it's time to make things happen with them using events. Fortunately, handling events in edit fields and picture fields falls into familiar territory. It's similar to the dialog handling routines for our program windows.

Let's start by examining the various field event types and see how a program should respond to each one. As we look at each event, we'll create a subroutine to handle the event. In this way, we'll create a library of field event handling subroutines that we can drop into any program and know that they'll work.

PROGRAM 66. Other window fields.

```
LOCAL FN BuildFindWindow
   ' ••• STATIC TEXT FIELDS
   xPos% = 60
   tmp$ = "Find:"
   EDIT FIELD #21, tmp$,(20,15)-(xPos%-5,31),_statNoframed, _rightJust
   ' ••• ACTIVE EDIT FIELDS
   EDIT FIELD #_searchFLD,"",(xPos%,15)-(320,31),_framedNoCR, _leftJust
END FN
LOCAL FN BuildPrintWindow
' ••• STATIC TEXT FIELDS
   EDIT FIELD #99 , "to", (210,50)-(230,65),_statNoframed, _centerJust
   ' ••• ACTIVE EDIT FIELDS
   tmp$ = STR$(gMaxRecords%)
   EDIT FIELD #_lastFLD, tmp$, (240,50)-(275,65),_framedNoCR,_centerJust
   tmp$ = STR$(gRecordNum% + 1)
   EDIT FIELD #_firstFLD, tmp$, (165,50)-(200,65),_framedNoCR,_centerJust
END FN
LOCAL FN BuildGotoWindow
' ••• STATIC TEXT FIELDS
   tmp$ = "Goto Record:"
   EDIT FIELD #99, tmp$, (20,15)-(135,31), _statNoframed, _leftJust
   ' ••• ACTIVE EDIT FIELDS
   tmp$ = STR$(gRecordNum%)
   tmp$ = RIGHT$(tmp$, LEN(tmp$) - 1)
   EDIT FIELD #_gotoFLD, tmp$, (135,15)-(160,31), _framedNoCR, _centerJust
END FN
```

Handling Mouse Clicks

Any mouse click within an active edit field generates an `_efClick`. This method is the most direct way of moving between several edit fields. Additionally, because we have changed the active field, we need to update the cursor to reflect it's position over the active field. The routine to handle `_efClick` events is shown in Program 67.

PROGRAM 67. Click handling events.

```
LOCAL FN EFClickEvent (fieldID%)
   EDIT FIELD #fieldID%
   CURSOR _iBeamCursor
END FN
```

Handling Tabs

One common way of moving between multiple edit fields is to use the Tab key. The Tab key normally moves the cursor to the next active edit field, while the Shift-Tab combination moves the cursor to the previous field. Both are identified using the `_efTab` and `_efShiftTab` constants.

When the user presses the Tab key, an `_efTab` event is generated. It's up to the program to trap this event in a dialog handler and respond appropriately.

This is where the sequential numbering of active fields comes into the game. Since all of the fields in the Data Entry window lie in the range of `_dbNameFLD` to `_dbFaxFLD` (1 - 8), it's child's play to write a routine that will handle this type of incrementing. Additionally, by passing the event type to the routine, we can also use it to handle Shift-Tab events. This dual purpose routine is shown in Program 68.

The `TabShiftTabEvents` subroutine expects three parameters: the event, the lowest field count to cycle through, and the highest. We need to pass these variables to ensure that any window that requires tabbing can use this same routine. When called, the subroutine examines the event and determines which course of action to follow.

PROGRAM 68. Handling Tab and Shift-Tab events.

```
LOCAL FN TabShiftTabEvents (dlgEvnt%, startFld%, lastFld%)
  LONG IF dlgEvnt% = _efTab
    fieldID% = (WINDOW (_efNum) MOD lastFld%) + 1
    IF fieldID% > lastFld% THEN fieldID% = startFld%
  XELSE
    fieldID% = (WINDOW (_efNum) - 1
    IF fieldID% < startFld% THEN fieldID% = lastFld%
  END IF
  EDIT FIELD #fieldID%
END FN
```

If it's a Tab event, it determines the current active `fieldID%` and increments it by one. It then checks to ensure that `fieldID%` doesn't exceed `lastFld%`. If it does, we reset it to `startFld%`.

If it's a Shift-Tab event, it again determines the current `fieldID%`, subtracts one from that, then checks to ensure it isn't below the `startFld%` value. If it is, it's set to `lastFld%`. Finally, the new active field is activated. Note that we use `EDIT FIELD` so that the text of the newly active field is completely selected.

In both cases you should note that we always wrap the Tabs and Shift-Tabs around the active fields in the window. By changing the values of `startFld%`

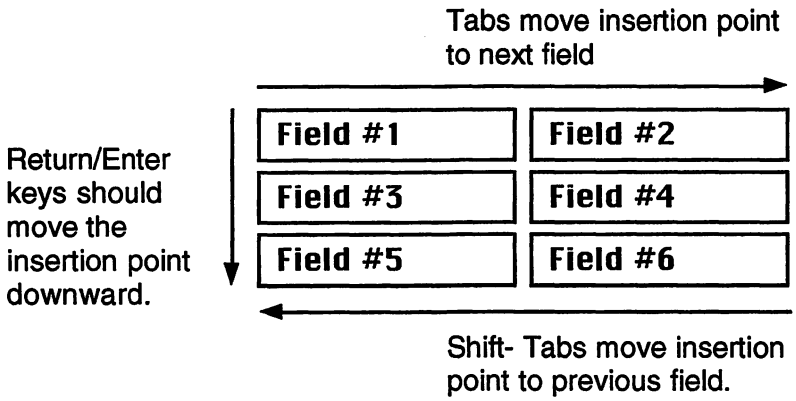
and `lastFld%`, we can use this routine for any window that requires tabbing support.

Return Keys

Another common method of switching edit fields is to use the Return key. This method only works if the fields used in the window are of the non-carriage return type, otherwise the key press will be intercepted by the active edit field and applied to the field itself.

In many cases, a program may have to handle Return keys separately from Tab key events. The illustration in Figure 40 shows one method of handling both Return and Tab keys.

FIGURE 40. Handling Return key events.



Unless you have some special processing, use `FN TabShiftTabEvents` to handle the Return key event. Just remember to pass an `_efTab` event instead of `_efReturn` to the subroutine.

We will use the Return key for other actions. Since most of our windows have a default shadow button, the normal action would be to activate the default button when the Return key is pressed. We can do this by intercepting the `_efReturn` event and converting it into a `_btnClick`. The easiest place to do this is right at the top of each window's dialog handler before the event gets passed onto the `SELECT/END SELECT` structure. The code to enter looks like this:

```
LONG IF _efReturn
  dlgID% = FN ChangeReturnToBtn (dlgEvnt%, btnID%)
END IF
```

where `btnID%` represents the window's default button `btnID%`. The subroutine `ChangeReturnToBtn` does exactly what it says. It accepts both the

dlgEvnt% and dlgID%. It then calls the HiliteSelectedButton subroutine to briefly invert the default button, then converts the dlgEvnt% from _efReturn to _btnClick. It does this using a simple trick. When passed the dlgEvnt%, it accepts not the event, but the address of the event, which it uses to replace _efReturn event with a _btnClick.

Given the address to anything, we can modify it by making it look like a record. We'll see how to use this technique when we talk about Records later.

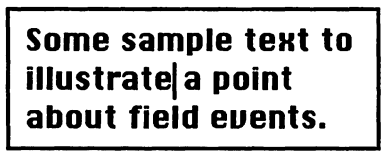
Now that the support routines are in place, add a call to ChangeReturnToBtn in the Data Entry, Find, Goto, and Print window dialog handling subroutines.

Arrow Keys

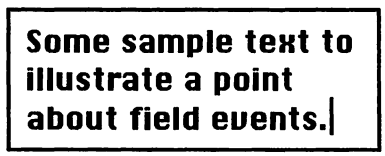
Arrow keys must also be handled in a special manner.

When the insertion point is within the text of a field, the field will suppress any arrow key event and move the insertion point as directed by the arrow key. For example, an up arrow will cause the insertion point to move up one line in the text. A left arrow key will move the insertion point one character left of its current position.

FIGURE 41. Handling arrow key events.



In this example, the insertion point is inside the text of an edit field. When the user presses an arrow key, the insertion point moves in the direction of the arrow key pressed.



However, if the insertion point is located at the end or beginning of the field's text, an event is generated when the user presses any arrow key.

When the insertion point is at the very beginning or end of the field's body text, the program will receive an arrow key event message. The program can then use the event to move the insertion point to another field entirely. In most cases, by passing the correct event type, you can call FN TabShiftTabEvents for all of your arrow events.

Peak Performance

We've only covered the fundamentals of using edit and picture fields. What follows are some additional tricks you might find useful.

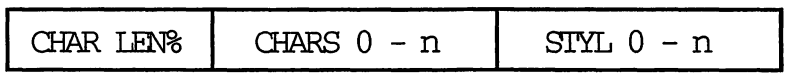
Using all 32K

None of the fields in our *SimpleBase* program required access to more than 255 characters. But since edit fields can contain up to 32,365 characters, there may come a time when accessing all of them is necessary.

ZTXT Method

The quickest method of accessing all 32K is to use GET FIELD. GET FIELD returns a handle (see the chapter "Resources"for more information on handles) containing a combination of both text and style information, commonly referred to as ZTXT. A ZTXT handle has the following format:

FIGURE 42. ZTXT data format.



Each ZTXT handle begins with an integer value that contains the character count, it's then followed by character data, and finally by any style information. So to get a ZTXT handle, do this:

```
GET FIELD zTxtH& , fieldID%
```

Once you have a ZTXT handle, it can be written to disk using:

```
WRITE FIELD #fileID%, zTxtH&
```

and read back into memory using:

```
READ FIELD #fileID%, zTxtH&
```

One thing to remember, is to always dispose of a ZTXT handle using KILL FIELD. Failure to dispose of this handle properly, either by forgetting to dispose of it or using another command, can create memory problems later.

Reading Text to a Field

Now that we understand the ZTXT format, let's look at a common problem, reading a TEXT file into an edit field. Most people come very close, but usually seem to end up with extra garbage characters at the end of the text. The reason is simple, they passed a handle containing nothing but text to the edit field. The field interpreted the first two characters as the number of characters, and read that many characters into the field. For example, say we had the following text in a handle:

```
Fred was here.
```

The statements EDIT FIELD and EDIT$ would see it like this:

```
<18034>ed was here.
```

Where the 18034 is the character count returned by the "Fr" characters. This is way too many characters than in our little example.

EDITFIELD and EDIT$ don't care. They will blindly accept this incorrect count and process beyond the handle into random memory. The result is garbage characters in the field.

PROGRAM 69. Text File to Field reader.

```
CLEAR LOCAL
LOCAL FN TextFile2Field (fieldID%)
  filename$ = FILES$ (_fOpen, "TEXT", , wdRefNum%)
  LONG IF LEN (filename$) > 0
    OPEN "I", #1, filename$, , wdRefNum%
    LONG IF SYESERROR = _noErr
      size% = LOF (1, 1)
      LONG IF size% < 32765
        hndl& = FN NEWHANDLE (size% + 2)
        LONG IF (hndl& <> 0) AND (SYSERROR = _noErr)
          hndl&..none% = size%
          osErr% = FN HLOCK (hndl&)
          READ FILE #1, [hndl&] + 2, size%
          osErr% = FN HUNLOCK (hndl&)
          EDIT$ (fieldID%) = &hndl&
          DEF DISPOSEH (hndl&)
        END IF
      END IF
      CLOSE #1
    END IF
  END IF
END FN
```

However, if the handle was reformatted with a leading integer it would look like this:

```
<14>Fred was here.
```

And everything would operate correctly. So the key is to add a length value at the beginning of the handle containing the text. The routine in Program 69 shows exactly how to do this. It checks for errors but doesn't give any notice if it encounters one, so be sure and beef it up before using it in your own programs.

Sending Field Text to a File

So, we got the text into the field, modified it as desired, now it's time to save it back out to disk. All we do is reverse the process that got it into the field. This is handled by the routine shown in Program 70.

PROGRAM 70. Field to Text File routine.

```
CLEAR LOCAL
LOCAL FN Field2TextFile (fieldID)
  GET FIELD txtHndl&, fieldID
  LONG IF txtHndl& = 0
    BEEP : BEEP
  XELSE
    txtLen% = {[txtHndl&]}
    osErr%  = FN HLOCK (txtHndl&)
    LONG IF osErr% = _noErr
      BLOCKMOVE [txtHndl&] + 2, [txtHndl&], txtLen%
      osErr = FN HUNLOCK (txtHndl&)
      LONG IF osErr = _noErr
        osErr = FN SETHANDLESIZE (txtHndl&, txtlen%)
        LONG IF osErr = _noErr
          tmp$ = "Save file as:"
          filename$ = FILES$ (_fSave,tmp$, untitled", wdRefNum%)
          LONG IF LEN(filename$) > 0
            DEF OPEN "TEXTxxxx"
            OPEN "O", #1, filename$, , wdRefNum%
            WRITE FILE #1, [txtHndl&], txtLen
            CLOSE #1
          END IF
        END IF
      END IF
    END IF
    KILL FIELD txtHndl&
  END IF
END FN
```

The subroutine takes a standard ZTXT handle, strips out the length and style data, and just writes the character data to disk as a TEXT file.

Cooldown

That was a long chapter. In it we learned all about edit and picture fields. We first learned about the various types of fields, their uses, and how to create them. Then, we repeated that for picture fields, emphasizing how to insert and retrieve data from both field types.

Next, we added fields to our windows and then learned about dealing with the multitude of events that are generated by user actions in a field. Finally, we learned how to read and write text files to and from edit fields.

Scroll Buttons

Warm-up

This chapter introduces a close cousin of the push, checkbox, and radio buttons, the scroll button. In this chapter you will learn:

- What scroll buttons are,
- The three types of scroll buttons,
- How to create scroll buttons,
- How to handle scroll button events, and
- How to link a scroll button with an edit field.

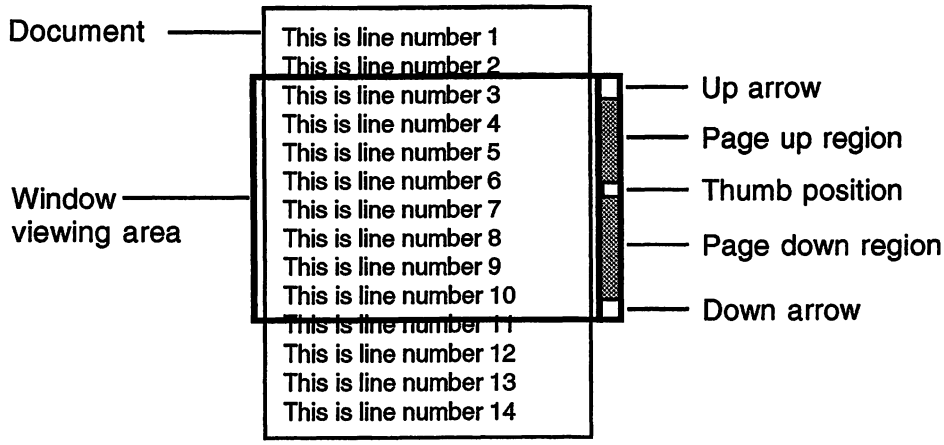
What are Scroll Buttons?

Scroll buttons are a variation of the standard control that enable you to view different sections of a document within a window. A scroll button represents an entire document in one dimension, either top to bottom, or left to right. By changing a scroll button's value, you can view different sections of the same document. As Figure 43 shows, if a document is larger than the window viewing area, you can use a scroll button to control the visible portion.

Scroll Button Features

Scroll buttons (also called scroll bars) consist of a rectangle with arrows at each end. Inside the rectangle is a square called the scroll box (or thumb). The remainder of the scroll button is known as the **gray area**.

FIGURE 43. How scrolling works.



Since the range of a scroll button is between -32,767 and 32,767, its current value is identified as the **scroll box value** (or thumb value). A click in a **scroll arrow** increments or decrements the current scroll value by one. Larger changes are made using a page value. A **page value** is assigned to the scroll button upon creation and can be changed later as needs dictate. A click in either gray area increments or decrements the scroll box by the page amount.

Creating Scroll Buttons

Creating scroll buttons is very similar to creating other buttons. They do require several more parameters than regular buttons since they are designed to support a range of values. These additional parameters include the: current value, minimum value, maximum value, and page value, as well as its location in the window and a type. The syntax to create a scroll button is:

```
SCROLL BUTTON #btnID%, current%, min%, max%, page%, rect, type%
```

A scroll button's `btnID%` resides in the same control list used by regular buttons. Therefore, a scroll button can't have a `btnID%` already used by another button in the window.

The `current%` setting is the position of the scroll box, and `min%` and `max%` represent the lowest and highest values `current%` can assume for the scroll button. The `page%` parameter controls the amount `current%` will change when the user clicks in a scroll button's gray area. The source shown in Program 71 (do not add to *SimpleBase*) demonstrates how to create different scroll buttons in a window.

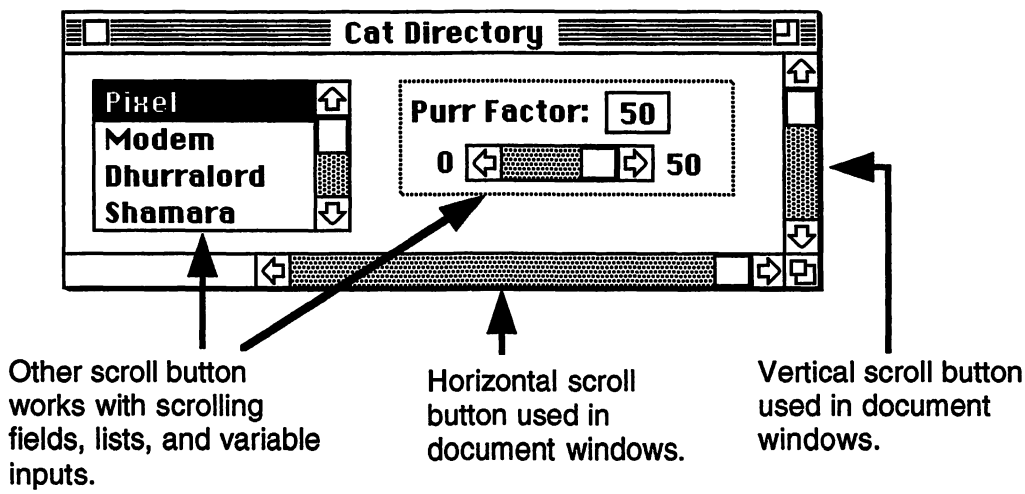
PROGRAM 71. Creating scroll buttons.

```
LOCAL FN BuildWnd
  WINDOW #1, "TEST", (0, 0) - (400, 300) , _doc
  SCROLL BUTTON #1, 1, 1, 100, 10, , _scrollVert
  SCROLL BUTTON #2, 1, 101, 200, 10, , _scrollHorz
  SCROLL BUTTON #3, 1, -100, 1, 10, , _scrollOther
END FN
FN BuildWnd
ON DIALOG FN HandleDialogEvent
DO
  HANDLEEVENTS
UNTIL 0
END
```

Types of Scroll Buttons

There are three variations of scroll buttons: `_scrollVert`, `_scrollHorz`, and `_scrollOther`. Vertical and horizontal scroll buttons are often associated with document windows. They are automatically placed at a window's edge by the runtime and remain there even when the window is resized. From their positions they control the vertical and horizontal scrolling of a window's contents. The type `_scrollOther`, or freeform scroll button, can be placed anywhere in a window and doesn't change position when the window is resized. It's useful for scrolling lists and edit fields. Examine Figure 44 to see examples of the three types of scroll buttons.

FIGURE 44. Scroll button types.



Getting the Scroll Box Value

We get the value of a scroll button using the BUTTON function. Scroll button events are reported by the _btnClick event. In a program's dialog handler use the dlgID% to identify which scroll button was clicked. The BUTTON function can return both the current and previous thumb positions. To get the current thumb position do this:

```
thisThumbPos% = BUTTON (dlgID%)
```

To get the previous thumb position use:

```
lastThumbPos% = BUTTON
```

Program 72 shows how to retrieve both thumb positions. As you can see, leaving off the btnID% returns the previous thumb position. Remember, this only works when a scroll button is clicked, not with other button types.

Setting the Thumb Value

You can reposition the thumb of a specified scroll button by setting the current% value of the scroll button. For example, set the thumb anywhere between the min% and max% values using:

```
SCROLL BUTTON #btnID%, current%
```

Other than the btnID% and the current% setting, no additional parameters are required.

PROGRAM 72. Getting the thumb position.

```
LOCAL FN HandleDialogEvent
  dlgEvnt% = DIALOG (0)
  dlgID% = DIALOG (dlgEvnt%)
  LONG IF dlgEvnt% = _btnClick
    LONG IF dlgID% = _myScrollBtn
      lastThumbPos% = BUTTON
      thisThumbPos% = BUTTON (dlgID%)
      SELECT dlgID%
        CASE 1 : PRINT "Vertical scroll…"
        CASE 2 : PRINT "Horizontal scroll…"
        CASE 3 : PRINT "Other scroll…"
      END SELECT
      PRINT "Last thumb position ="; lastThumbPos%
      PRINT "This thumb position ="; thisThumbPos%
    END IF
  END IF
END FN
```

Changing Scroll Button Values

Since a scroll button's scroll box represents a variable range of values, you may occasionally need to reset those values. For example, a document used in a word processing program constantly increases as the user enters more lines of text. If the scroll button doesn't change its max% value, scrolling to the bottom of the document would be impossible.

You can change most scroll button parameters. Exceptions are the btnID%, and the scroll button type which can't be changed once they're built. Everything else is fair game. Just specify the scroll button to change, and then set the new parameter. For example, to change the max% value, do this:

```
SCROLL BUTTON #btnID%, , , max%
```

Set a new page% value like this:

```
SCROLL BUTTON #btnID%, , , , page%
```

And to reset new min% and max% values:

```
SCROLL BUTTON #btnID%, , min%, max%
```

Note that all missing parameters are left unaffected by the changes. To disable a scroll button, set all parameters to zero like this:

```
SCROLL BUTTON #btnID%, 0, 0, 0, 0
```

Linked Scroll Buttons

One of the best features of FB is its ability to link a scroll button with an edit field to produce a scrolling edit field. With just two lines of FB code you accomplish what other languages take dozens, if not hundreds, of lines to do.

There are a few requirements to make this happen successfully, including:

- You must use the styled edit field exclusively.
- The edit field and the scroll button must use the same ID value, i.e. btnID% = fieldID%.
- Define a linked scroll button with the negative sign in both the EDIT FIELD and SCROLL BUTTON statements.

That's all the restrictions. So, to create a linked scrolling field, merely do this:

```
EDIT FIELD #-1, tmp$, rect, type, just
SCROLL BUTTON #-1, , , , , rect, type
```

Note that both the edit field and the scroll button have identical negative IDs. The runtime uses this to determine which field will be linked to which scroll button. If the scroll bar is of the type _scrollVert or _scrollHorz, you don't even have to specify the thumb position, the runtime handles that also.

When linking a field and a `_scrollOther` scroll button type, you must perform some calculations to properly position the scroll button next to the edit field. There is a small subroutine called `BuildScrollFld` in the FB Library help file that performs these calculations for you.

Once a linked field is defined in a program, you don't have to handle a thing except for placing new data into the field. The runtime handles the details of keeping the edit field contents and scroll button's thumb in perfect sync.

Regular Exercise

Now that we understand scroll buttons a bit better, it's time to add one to *SimpleBase*. The main entry window doesn't require a scroll button, but the Help window is perfect. The Help window provides users with detailed instructions using a linked edit field and scroll button. Insert the code in Program 73 into the `FN BuildHelpWindow` routine.

PROGRAM 73. Build Help window.

```
LOCAL FN BuildHelpWindow
  tmp$ = "SimpleBase Help"
  WINDOW #-_helpWIND, tmp$, (0,0)-(400,260), _doc, _helpWIND
  TEXT _sysFont, 12
  ' ••• STATIC TEXT FIELD
  EDIT FIELD #-_helpFLD,"",(4,4)-(382,244), _statFramed, _leftJust
  SCROLL BUTTON #-_helpSCROLL,1,1,1,10,, _scrollVert
END FN
```

Once the Help window build routine is complete, finish the `FN DialogHelpWindow` by adding the event handling code shown in Program 74. Here, the goal is to watch for a `_wndRefresh` event. If we get one, we need to ensure that the edit field is resized so that it fills the window. We do that by recalculating the window size and setting the edit field's rectangle properly.

We'll see how to add the program instructions later when we talk about `TEXT` resources in the chapter "Strings & Text", but for now, the Help window is set up to handle them once they become available.

Peak Performance

Working with scroll buttons is exactly the same as working with regular buttons. Although they share the same control record, some portions of it are used differently.

PROGRAM 74. Revised Help window dialog handler.

```
LOCAL FN DialogHelpWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndRefresh
      wndX = WINDOW (_width) : wndY = WINDOW (_height)
      EDIT FIELD #_helpFLD, "", (4,34)-(wndX-4,wndY-4)
      PLOT 0, 30 TO wndX, 30
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _prevHelpBTN
          DEC (gHelpID%)
          IF gHelpID% < _firstHelp THEN gHelpID% = _lastHelp
        CASE _nextHelpBTN
          INC (gHelpID%)
          IF gHelpID% > _lastHelp THEN gHelpID% = _firstHelp
        CASE ELSE
      END SELECT
      LONG IF dlgID% > _helpSCROLL
        SCROLL BUTTON #_helpSCROLL, 1
        EDIT FIELD #_helpFLD, %gHelpID%
      END IF
    CASE ELSE
  END SELECT
END FN
```

Scroll Indicators

Occasionally, when using a scroll button, you may wish to display the current value. One means of doing this is to find the location of the thumb and display the `current%` setting within it. The example in Program 75 shows how to access the thumb position within a control record.

The thumb position is stored as a handle within the `contrlData` field of the control record. We start by creating a window with two scroll buttons, then enable dialog events. When a `_btnClick` event is detected, the routine gets the control's handle using `BUTTON&`, then uses it to access the `contrlData` handle. Once it has the handle, it looks two bytes into the handle and locates the rectangle position of the thumb in the window. A quick copy to a local rectangle and draw the thumb value inside the thumb position in the window. Then, the `AUTOCLIP` statement disables all clipping of the control area and the `current%` setting is drawn at the thumb's position. `AUTOCLIP` is re-enabled and the dialog handler ends.

PROGRAM 75. Scroll indicators example.

```
LOCAL
DIM rect.8
LOCAL FN DoDialog
  dlgEvnt% = DIALOG(0)
  dlgID% = DIALOG (dlgEvnt%)
  LONG IF dlgEvnt% = _btnClick
    scrollValue% = BUTTON (dlgID%)
    AUTOCLIP 0 : TEXT ,7,,1
    btnH& = BUTTON& (dlgID%)
    rect;8 = [btnH&..contrlData&] + 2
    PRINT% (rect.left%, rect.top% + 10) USING "###";scrollValue%;
    LONG IF dlgID% = 1
      tmp$ = "Vertical scroll ="
    XELSE
      tmp$ = "Horizonal scroll ="
    END IF
    TEXT ,12,,0 : PRINT%(10,20) tmp$;scrollValue%;
    CLS LINE : AUTOCLIP 1
  END IF
END FN
LOCAL FN Init
  WIDTH _noTextWrap
  WINDOW 1, "SCROLL BUTTON", , _doc : TEXT _geneva, 9, ,0
  SCROLL BUTTON #1, 1, 1, 100, 10,, _scrollvert
  SCROLL BUTTON #2, 1, 1, 100, 10,, _scrollHorz
END FN
FN Init
ON DIALOG FN DoDialog
DO
  HANDLEEVENTS
UNTIL 0
END
```

- *Note, the FB runtime automatically clips out of the drawing area any space occupied by a button or field. You can use* `AUTOCLIP` *when needed or set the window attribute* `_noAutoclip` *when building the window.*

Cooldown

That's it for scroll buttons. In this chapter you have learned what scroll buttons are, the three types available to us, how to create them, and how to change their values including `current%`, `min%`, `max%`, and `page%`. You also learned how to link a scroll button with an edit field.

CHAPTER 10

Records

Warm-up

We've done the interface work, now it's time to begin manipulating the data we need to store in our employee files. One way to do that is to use record structures. In this chapter you will learn:

- What records are,
- How to define records and create record variables,
- How to assign and retrieve data from record variables,
- How to create record arrays,
- How to move records to edit fields and back, and
- How to write and read records from a disk file.

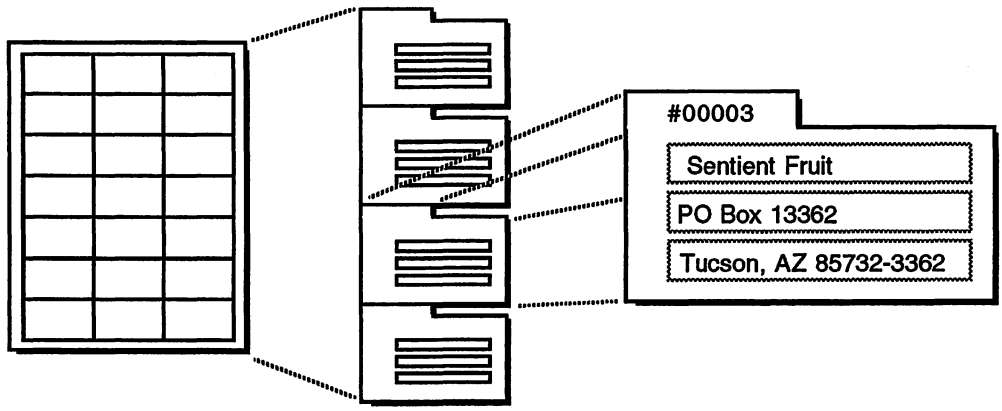
What are Records?

Records are a method of organizing related pieces of data into a structure that can be manipulated as a single object. A **record** is a structured format for data. Records can store any type of data including text, values, and graphical information.

A record is made up of separate fields. A **field** is a component of a record that contains data in defined format. For example, a rectangle record consists of four fields (top, left, bottom, right), each of which holds an integer value. Each rectangle field requires two bytes to maintain its data, resulting in a total of eight bytes for the rectangle record. The combined size of all record fields is called the **record size**.

When accessing a record, it's usually referred to by a record number. The **record number** is the indexed location of the record within a file. Record numbers normally begin with zero and increment by one for each record thereafter.

FIGURE 45. Program data file containing records.



- *Note, setting "Arrays without element zero" in the Preferences can change this.*

Record Sizes

An individual record can be up to 32K in size. Records used in arrays, however, are restricted to 256 bytes due to runtime limitations. Future versions of FB will probably not have this restriction.

- *The 32K limit is imposed by the absolute size of single code blocks. Future machines may not have this limitation.*

Defining Records

There are three statements required to define a record structure: `DIM RECORD`, `DIM`, and `DIM END RECORD`. `DIM RECORD` starts the record definition, `DIM` defines the individual fields within the record, and `DIM END RECORD` defines the record's end and returns its total size and type. A record defined using these commands becomes a variable type just as integer, long integer, string, single–, and double-precision are variable types.

One common record is the rectangle record. It's used to define windows, buttons, edit fields, graphics, and other object positions. A rectangle record contains four integers specifying the top, left, bottom, and right coordinates. We define a rectangle record like this:

```
DIM RECORD rect
   DIM gtop%
   DIM gleft%
   DIM gbottom%
   DIM gright%
DIM END RECORD .rectRecSize
```

As you can see, each field in the record follows the other in the definition. This is exactly how the variables are placed into memory, one right after the other. The last statement, `DIM END RECORD`, returns the total size of the newly defined record. The use of the dot (.) in the statement before the size variable creates the record structure but allocates *NO* memory for the record itself.

- *By convention, I always define my record sizes to end with the suffix `RecSize` for easier recognition.*

Record Types

Why separate the record definition from the record? It's mainly for clarity and consistency in implementation. You don't see or use an integer, you use variables of the integer type. Likewise, you don't have a string to manipulate, you have a variable of the type string. This same method should be used with records. Define the record type, then define variables of that type. For example, a look at other variable types reveals this general variable format:

```
<varName>[varType]
```

where the `varType` is always an optional identifier. A look at the different variable types shows they all run true to form:

```
<integerVar>[%]
<longIntVar>[&]
<stringVar>[$]
<singlePrecVar>[!]
<doublePrecVar>[#]
```

whereas a record variable looks like this:

```
<recordVar>.recordType
```

With the dot separating the variable from the record type. When defining record variables, the record type must always be appended to the definition.

Record Allocation

Using the same example, we can define multiple variables of the same record type like this:

```
DIM aRect.rectRecSize
DIM bRect.rectRecSize
DIM cRect.rectRecSize
```

Where `rectRecSize` represents the record type, just as the shorthand modifiers `%`, `&`, `$`, `!`, and `#` represent their respective variable types.

However, there may be occasions where this separation of record type and variable is inconvenient or unwieldy. In that case, you can use an underscore instead of a dot in the `DIM END RECORD` statement like this:

```
DIM RECORD rect
  DIM gTop%
  DIM gLeft%
  DIM gBottom%
  DIM gRight%
DIM END RECORD _rectRecSize
```

This sets aside memory for the record as it's defined. You now have both the structure and the variable in one bundle. References to it are made using the record name (`rect`) like this:

```
rect.gTop% = 10
```

References to individual fields can be by field name or record offset. For example, I could say:

```
gTop% = 10
```

or I could say:

```
rect.gTop% = 10
```

Both of these set the `field gTop%`. I prefer the second method because it identifies which record I am using. If I just use `gTop%`, it might as well be defined as a global variable.

Variable Sizes

There is a shorthand method of defining adjacent variables. Instead of defining each one separately, define them instead as a single group. Since we know the size of each field variable (2 bytes), it's easy to calculate the total record size of eight bytes. If we defined a record to hold car information, it might look like this:

```
DIM RECORD car
  DIM carRect.8
  DIM carModel%
  DIM carStyle%
DIM END .carRecSize
```

The use of the dot (.) in the `DIM` statement tells the runtime to define the variable using the length after the dot and remember it as a record. As we'll see later, defining a variable as a record has some benefits, including the ability to copy one record variable into another.

We could have also defined the variable like this:

```
DIM RECORD car
  DIM carRect;8
  DIM carModel%
  DIM carStyle%
DIM END .carRecSize
```

The use of the semi-colon (;) after the variable overrides the normal variable size and sets it to the specified value. Instead of being a sub-record within the record, we have a variable `carRect%` with an additional six bytes attached to it. I don't recommend using this format except for special cases. You gain nothing by using it, and lose all record handling capabilities allowed with the dot specifier.

Zero Length Variables

One good use for the semi-colon is to define a variable of zero length. Many times you may need to refer to a specific variable in two different ways. The common occurrence of this is with a mouse point.

A mouse point consists of two integer variables specifying the vertical and horizontal position of the cursor. Some Toolbox routines, however, expect you to pass the point as a single long integer. How can we do both? Simple- create a long integer variable of zero length followed by two integer variables like this:

```
DIM where&;0, posY%, posX%
```

The variable `where&` occupies the same memory as `posY%` and `posX%` (remember, memory is assigned contiguously) because it's defined as having no length. The runtime happily assigns the subsequent `posY%` and `posX%` variables the same memory location in its variable lookup table. An example of this is:

```
posY% = 99
posX% = 88
```

So, if you need to pass the mouse position as a long integer you can pass `where&`, or access either the vertical or horizontal positions using `posY%` and `posX%`.

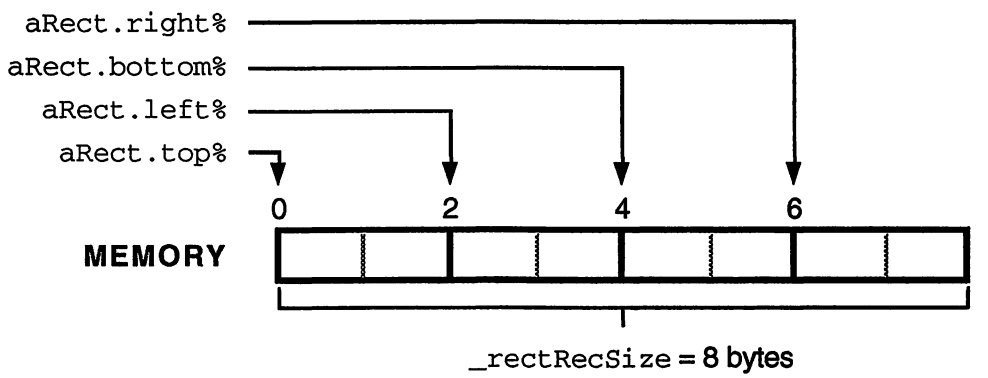
Accessing Record Data

Now that we understand how to create record types, let's look at storing and retrieving data in a record variable.

When you create a record structure, each field name used in the record becomes a constant. Remember that constants are predefined values that can be used in place of variables. When we created the rectangle record earlier it

created a total of five constants, one for each field name (_gTop, _gLeft, _gBottom, and _gRight), and one for the record size (_rectRecSize). Each field constant represents the actual byte offset from the beginning of the record. Thus, _gTop = 0, while the field constant _gBottom = 4. The illustration in Figure 46 shows how the record offsets are used to point to the individual field data in the variable aRect.

FIGURE 46. Record storage in memory.



When a variable is created by a program, the runtime links the variable name to a particular address in memory. Any references in the program to that variable causes the runtime to look up the name in its variable list and locate the address where the variable's data is stored.

A record variable consists of a sequence of contiguous bytes stored at a specific memory location. If a constant is added to the variable, the runtime seeks out the variable's memory location, then offsets the search from that location by the number of bytes specified by the constant. For example, to print the gbottom% field in a rectangle record you could do this:

```
PRINT aRect.gbottom%
```

The runtime looks up the variable aRect and locates its memory address. Then, it adds the constant value _gbottom (4) to the first address. The single dot (.) tells the runtime to add the subsequent constant to the base variable's memory address found in the lookup table. Finally, it grabs the number of bytes specified by the type identifier (%) and prints the information. It sounds more complex than it is and besides, the runtime is doing all the work anyway. In reality, you don't have to worry about all this.

Assigning Record Data

You assign values to the fields in a record just as you would assign any other variable value. The only difference is that you use a field constant to provide

the offset into the record where the data should be stored. Additionally, you must supply a variable type identifier (%, &, $, !, #) to tell the runtime exactly how much data it should store in the field.

Let's assume we've defined a record structure that looks like this:

```
DIM RECORD miscStuff
  DIM myVar1%
  DIM myVar2&
  DIM myVar3$
DIM END RECORD .miscRecSize
```

And created a variable of that record type like this:

```
DIM myRecord.miscRecSize
```

We can use the implied (with the equal sign) `LET` statement to assign values to `myRecord` like this:

```
myRecord.myVar1% = 10
myRecord.myVar2& = &HAABB
myRecord.myVar3$ = "Fred Hott"
```

Which stores an integer (%) into `myVar1`, a long integer (&) into `myVar2`, and a string ($) into `myVar3`.

Retrieving Record Data

Retrieving data stored in a record variable is just as easy. Using the same format shown in the previous section, specify the record name, the field constant with its dot, and a type identifier. To retrieve the information stored in the above example do this:

```
PRINT myRecord.myVar1%
PRINT myRecord.myVar2&
PRINT myRecord.myVar3$
```

which would output:

```
10
&HAABB
Fred Hott
```

Note again the use of the type identifiers (%, &, $, !, #) in the reading of the field data. These are required to tell the runtime how it should read the data in the record.

Nested Records

One of the best features of record types is the ability to nest one record structure inside another. For example, if you had an object record that requires a rectangle, you could define it as shown in Program 76.

PROGRAM 76. Nesting record types.

```
DIM RECORD rectRec
  DIM rect%.8
DIM END RECORD .rectRecSize
DIM RECORD objectRec
  DIM oNum%
  DIM oFlag%
  DIM oRect.rectRecSize
DIM END RECORD .objectRecSize
```

Next, create a variable of type `objectRecSize` and assign values to the rectangle record inside as shown in Program 77.

Additionally, you can use any of the standard Toolbox procedures designed to create specific data structures. Here is another way to assign rectangle information:

```
CALL SETRECT (gObj.oRect, 10, 10, 120, 120)
```

As you can see, the runtime knows where the `gObj` variable is located in memory. It then uses the field constant (`oRect`) to access the nested record, and the rectangle constants to reach the correct field. To print out the bottom field again you would do this:

```
PRINT "Bottom = ";gObk.oRect.bottom%
```

PROGRAM 77. Assigning record field values.

```
DIM gObj.objectRec
gObj.oNum%= 1
gObj.oFlag%= &HAA
gObj.oRect.top%= 10
gObj.oRect.left%= 10
gObj.oRect.bottom%= 120
gObj.oRect.right%= 120
```

Power Records

One reason for using records is the easy integration it has with all of the predefined record structures built into the Macintosh. A simple run through the *Inside Macintosh* volumes shows hundreds of record types just begging to be used, if only you could get to the information quickly and easily. Now you can.

For example, every window contains a record that holds all of the information required by that window. FB gives us a pointer to the window record using either the `WINDOW` function or the `GET WINDOW` statement like this:

```
wndPtr& = WINDOW (_wndPointer)
```

or this:

```
GET WINDOW #wndID%, wndPtr&
```

Before constants were available, once you had a window pointer it was difficult to access or change the fields directly. The methods included reproducing the record using DIM statements or using PEEKs and POKEs to read/change the data.

Now, as soon as you have any window pointer, use the Toolbox field names as offsets into the record structure. To access information from the window record, use its own field constants (see *Inside Macintosh* or use the Constants tool) as shown in Program 78.

PROGRAM 78. Reading record fields.

```
PRINT wndPtr&.txFont%
PRINT wndPtr&.txSize%
PRINT wndPtr&.fgColor&
PRINT wndPtr&.bkColor&
PRINT wndPtr&.windowKind%
PRINT wndPtr&.strucRgn&
PRINT wndPtr&.contRgn&
PRINT wndPtr&..titleHandle$
```

Notice the use of double dots for the window title. Use this technique to read the contents of a handle (a handle is a pointer to a pointer) stored in a record. In this case, it's the window title. Of course, you can also assign values directly using this same technique. Now, isn't that better?

Copying Records

Another nifty record feature is the ability to copy data from one record into another, as long as both are identical record types. For example, you can copy one rectangle variable into another by doing this:

```
aRect = bRect
```

Note that you can't do this if the variables don't have matching record types, nor if they've been defined using a semi-colon.

Record Arrays

Record arrays are easy to implement and use. Taking our earlier example of ten rectangles we can define an array to store them like this:

```
DIM RECORD rectRec
  DIM rect%.8
DIM END RECORD .rectRec
```

```
DIM gMyRect.rectRec (9)
```

and use it like this:

```
elementNum% = 0
CALL SETRECT (gMyRect(elementNum%), 10, 10, 120, 120)
```

We then define constants to represent individual rectangles and access them this way:

```
_wndR = 0
_okBtn= 3
CALL FRAMERECT (gMyRect(_wndR))
CALL FRAMERECT (gMyRect(_okBtn))
```

Records used in arrays are limited to 256 bytes in size. Using records larger than this can cause unpredictable results when reading or writing to records in the array.

Reading & Writing Records to Files

Variables defined as record types can be written to and read from disk files by using their names. Since the runtime already knows the total record size, it's not much of a chore. For example, to write the object record defined earlier to disk you can do something like what's shown in Program 79.

PROGRAM 79. Writing records to disk.

```
LOCAL FN WriteRecord (recNum%)
  OPEN "R",1,"MyFile", _objectRec, wdRefNum%
  RECORD #1, recNum%, 0
  WRITE #1, myObj
  CLOSE #1
END FN
```

This routine opens a file with the `OPEN` statement, sets the record position using `RECORD`, then writes the record variable to disk with `WRITE`. Finally, the file is a closed. We will examine these routines in more detail later in the chapter "Files".

To read it back into memory just reverse the process and use a routine much like that shown in Program 80.

PROGRAM 80. Reading records from disk.

```
LOCAL FN ReadRecord (recNum%)
  OPEN "R",1,"MyFile", _objectRec, wdRefNum%
  RECORD #1, recNum%, 0
  READ #1, myObj
  CLOSE #1
END FN
```

Again, what could be easier?

Regular Exercise

Now that we understand records better, it's time to develop the ones required by *SimpleBase*. The record structure for our project can be seen in Program 81.

In most cases the fields in this record are string variables. This was done because most of the information we will be storing is best described by strings, and not numbers. The two exceptions relate to the employee's department which we will represent using a button number, and the resource ID of the employee picture.

The semi-colon variation of specifying a variable size was used here to define our string fields. This tells the runtime that we want to set aside X number of bytes for that variable, no more, no less. This means we must be careful to check that the data we write to those fields never exceeds the length count, or risk corrupting the variable adjacent in memory. Also, it means our actual string length is one character less than the formatted size, leaving room for a length byte.

PROGRAM 81. SimpleBase record structure.

```
' --- RECORDS ---------------------------------------
DIM RECORD dbRecord
   DIM dbName$;64
   DIM dbAddr$;64
   DIM dbCity$;32
   DIM dbMyState$;4
   DIM dbZip$;12
   DIM dbPhone$;12
   DIM dbFax$;12
   DIM dbDeptNum%
   DIM dbPictID%
   DIM dbExtra&
DIM END RECORD .dbRecordSize
```

Records to Fields and Back

Once we've defined the record structure, we need some way of storing that record info. And after the records are stored in memory, we need some way to read the data into our edit fields and retrieve any changes for later updating.

When we first talked about records, we stated that defining a record structure is not the same as defining a record variable. Once the structure is in place,

we define a record variable using the record type (record size). For our program we only need one record variable which we define like this:

```
DIM gEmployee.dbRecordSize
```

Where DIM RECORD only defined the record structure, this DIM statement now sets aside memory for the employee record. This is the location where the current active record's information will be stored.

Not to jump ahead of ourselves, but if we assume that the gEmployee variable contains valid employee information, we need some means of transferring that data into the relevant edit fields of the data entry window for display. The subroutine RecordFieldToEF in Program 82 uses the field constants created earlier to place the data into the specified edit field.

PROGRAM 82. Record to edit field.

```
LOCAL FN RecordFieldToEF
  oldWnd% = WINDOW (_outputWnd)
  WINDOW OUTPUT #_dbEntryWIND
  EDIT$(_dbNameFLD)     = gEmployee.dbName$
  EDIT$(_dbAddrFLD)     = gEmployee.dbAddr$
  EDIT$(_dbCityFLD)     = gEmployee.dbCity$
  EDIT$(_dbStateFLD)    = gEmployee.dbMyState$
  EDIT$(_dbZipFLD)      = gEmployee.dbZip$
  EDIT$(_dbPhoneFLD)    = gEmployee.dbPhone$
  EDIT$(_dbFaxFLD)      = gEmployee.dbFax$
  EDIT$(_dbPhotoFLD)    = %gEmployee.dbPictID%
  FN RadioButtonHandler (_programBTN, _officeBTN, gEmployee.dbDeptNum%)
  WINDOW OUTPUT #oldWnd%
END FN
```

The subroutine starts by ensuring that we're transferring data from the correct window using. The WINDOW function returns the current output window, then WINDOW OUTPUT makes sure we're at the Data Entry window. When finished with the data transfer, it resets the original output window.

Next we use the EDIT$ statement to replace the contents of the specified edit or picture field with new data. There are two exceptions, the first is the RadioBtnHandler routine. It uses the value stored in dbDeptNum% as the department number to set the appropriate radio button. The second exception is the employee picture that is assigned directly, using the picture's resource ID.

Once the data is in the correct edit fields, it's possible (actually pretty darn likely) that the user will modify that information. Therefore, our next step

involves adding a routine to extract the new information in the proscribed fields and place them in the correct gEmployee record fields.

Remember, earlier we defined our string fields to be a finite length. That means we risk overwriting an adjacent field's data should one field exceed its length. To solve this, we add the subroutine CheckFieldLength$ that uses the LEFT$ statement to ensure that the data taken from an edit field doesn't exceed its field length. This routine is shown in Program 83.

PROGRAM 83. String length checker.

```
LOCAL FN CheckFieldLength$ (fieldID%, maxLen%)
  tmp$ = EDIT$ (fieldID%)
  LONG IF len (tmp$) > maxLen%
    BEEP
    tmp$ = LEFT$ (tmp$, maxLen%)
  END IF
END FN = tmp$
```

With our length checking routine in place, it's time to extract the data from the edit fields and copy it into our record. The EFtoRecordField function shown in Program 84. Note that the string length passed to CheckField-Length$ is one less than the string's defined length.

PROGRAM 84. Edit field to record.

```
LOCAL FN EFtoRecordField
  oldWnd% = WINDOW (_outputWnd)
  WINDOW OUTPUT #_dbEntryWIND
  gEmployee.dbName$    = FN CheckFieldLength$ (_dbNameFLD, 63)
  gEmployee.dbAddr$    = FN CheckFieldLength$ (_dbAddrFLD, 63)
  gEmployee.dbCity$    = FN CheckFieldLength$ (_dbCityFLD, 31)
  gEmployee.dbMyState$ = FN CheckFieldLength$ (_dbStateFLD, 3)
  gEmployee.dbZip$     = FN CheckFieldLength$ (_dbZipFLD, 11)
  gEmployee.dbPhone$   = FN CheckFieldLength$ (_dbPhoneFLD, 11)
  gEmployee.dbFax$     = FN CheckFieldLength$ (_dbFaxFLD, 11)
  WINDOW OUTPUT #oldWnd%
END FN
```

Now we can deal with the string data itself. The department data is handled in the DialogEntryWindow routine when handling button events. Whenever a button is pushed in the Data Entry window, the program determines which button it was and responds by calling RadioBtnHandler and then setting the dbDeptNum% field. This is shown in Program 85. We will see how to handle the picture resource ID in a later chapter.

PROGRAM 85. Handling radio buttons.

```
SELECT evnt
  CASE _btnClick
    SELECT dlgID%
      CASE _newRecBTN
      ' skip showing others
      CASE ELSE
        FN RadioButtonHandler (_programBTN, _officeBTN, dlgID%)
        gEmployee.dbDeptNum% = dlgID%
    END SELECT
  CASE ELSE
END SELECT
```

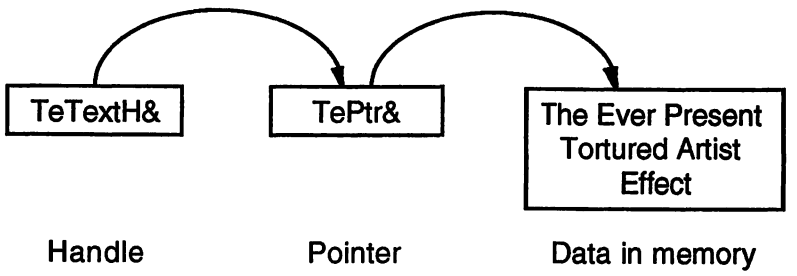
Peak Performance

While the implementation of records in FB was a welcome addition to BASIC programmers, it doesn't provide all of the features some people desire. Probably the main capability records currently lack, is that of including arrays within a record definition. While a problem, it does have a solution, one that allows you to have your arrays in a record. To do that, we'll borrow a strategy used quite often by Apple when implementing various Toolbox records. That strategy is handles.

Examine the Toolbox text edit record shown in the *FB Reference* manual under `TEHANDLE`. Note the record field `teTextH&`. This handle points to the actual text data for that field. Examine Figure 47 to see how the handle in the TE record points to a memory address. The data isn't stored in the record itself, only its handle address. We can use this same technique to create and manipulate arrays for our own records. Let's see how to do that.

To begin, we'll need three routines to deal effectively with our arrays: one to create the array, one to resize as required, and one to dispose of it once we're done. Also, we'll want our routine to work with any type or size of array, so it

FIGURE 47. From handle to data memory path.



must be flexible. Additionally, we need one little trick to pull off this array masquerade, and that is the XREF@ statement. XREF@ will link our amorphous handle to a more standard array structure.

With those goals in mind, let's see how to create records that contain arrays.

Creating the Array

The size of an array depends on two conditions, the number of elements in the array, and the size of each element. We should have some method of defining the standard sizes. The following is a list of the major variable types defined as constants, along with their default sizes in bytes:

```
_integer    = 2
_longInt    = 4
_single     = 4
_double     = 8
_string     = 256
```

- *Note that the* _single *and* _double *size specifications are dependent on their length settings in the Preferences dialog in FB.*

Next, define a small record structure to handle the array overhead required by our routines. It's not much, merely 10 bytes per usage, but very important to controlling the array. The record contains three fields. The first field is the array handle itself. The second contains the size of the array elements, be it integer, long integer, or other. Finally, we end the record with the total count of elements in the array. Our array record is shown in Program 86 as well as the record variable.

With all that out of the way, it's time to write a subroutine to create an array handle. The routine shown in Program 87 expects three variables: a pointer to

PROGRAM 86. Array handling record structure.

```
DIM RECORD array
  DIM elemSize%
  DIM numElems&
  DIM recArrayH&
DIM END RECORD .arrayRecSize
DIM RECORD test
  DIM testRect.8
  DIM testArray.arrayRecSize
  DIM 63 testStr$
DIM END RECORD .testRecSize
DIM gTest.testRecSize
END GLOBALS
```

PROGRAM 87. Create record array.

```
LOCAL FN CreateArray (@recFieldPtr&, size%, elements&)
   recFieldPtr&.elemSize% = size%
   recFieldPtr&.numElems& = elements&
   recFieldPtr&.myArrayH& = FN NEWHANDLE (elements& * size%)
   LONG IF (recFieldPtr&.myArrayH& = 0) OR (SYSERROR <> 0)
      BEEP : BEEP
      arrayErr% = SYSERROR
   END IF
END FN = arrayErr%
```

a record field, the size of the elements (in bytes) in the array, and the number of elements the array should initially contain.

The `FN CreateArray` routine starts by assigning the `size%` and `elements&` parameters to the specified record field. It then creates a handle of the calculated size (`size% * elements&`) using the Toolbox function `NewHandle`. If successful, we get a valid handle back which is stored in the correct field of the record. If the array handle comes back zero, or a system error is detected, we set an error and exit the function. Always test the error result returned by this function. If the error code is anything but zero, the handle wasn't created. Any subsequent attempt to use it will fail dismally, and probably with assorted pyrotechnics.

- *I can't emphasize this point enough. When using handles and pointers on the Macintosh, always test for valid parameters before continuing with the program. You'll avoid a lot of programming trouble if you get into the habit of always checking them.*

We call the routine like this:

```
err% = FN CreateArray (gTest.testArray, _integer, 100)
```

Where `gTest.testArray` is the field in the record that needs an array. We want integers (2 bytes each) so pass that constant, and 100 elements makes for a nice round handle of 200 bytes.

Linking the Array

Once we have a valid handle in our record, the next step is to link it to a common array. We do that using `XREF@`. `XREF@` acts like a translator, linking the defined array structure to the named handle. In this way, we don't have to calculate offsets into the handle to set or get information, `XREF@` does it for us. Remember, `XREF@` is a close cousin to `DIM`, so we use the same syntax we would with `DIM` to define the array structure. We do that like this:

```
XREF@ gTest.testArray.recArrayH% (100)
```

Where we use the same name as in the handle to define an integer array with 100 elements. Note the use of the % symbol to specify an integer array, just as you would do with a DIM statement. Also note that we must use the same name as the handle in order for the runtime to make the link between the array and the handle.

Once the array and handle are linked, we can fill in and read values as if it were a regular array. To fill in the 100 elements and see them you do this:

```
FOR count = 1 TO 100
  gTest.testArray.recArrayH% (count) = count
  PRINT "Array: "; gTest.testArray.recArrayH% (count)
NEXT count
```

As soon as data is in the array, use it like any other array structure in the program. You can change values like this:

```
gTest.testArray% (23) = 333
```

Use arrays in calculations, or anywhere else your program requires them. Remember, it's just an array, use it like one.

Disposing of the Array

When finished with the array we must dispose of the handle used to store the data. This breaks the link to the XREF@ array structure and frees up the memory occupied by the handle for other uses. The routine to dispose of the handle is shown in Program 88.

Again, we ensure we have a valid handle before trying to dispose of it, then reset the handle to zero so it can't be reused again.

PROGRAM 88. Disposing of record arrays function.

```
LOCAL FN DisposeArray (@recFieldPtr&)
   LONG IF recFieldPtr&.myArrayH& = 0
      arrayErr% = _nilHandleErr
   XELSE
      DEF DISPOSEH (recFieldPtr&.myArrayH&)
      recFieldPtr&.myArrayH& = 0
   END IF
END FN = arrayErr%
```

Resizing the Array

Now we come to another benefit of using a handle to create an array. By resizing a handle we can effectively increase or decrease the number of elements the array structure has access to. That means that as your

PROGRAM 89. Resizing record arrays function.

```
LOCAL FN SizeArray (@recFieldPtr&, newElemCount&)
  LONG IF recFieldPtr&.myArrayH& = 0
    arrayErr% = _nilHandleErr
  XELSE
    newSize& = recFieldPtr&.elemSize% * newElemCount&
    oldSize& = FN GETHANDLESIZE (recFieldPtr&.myArrayH&)
    LONG IF  newSize& <> oldSize&
      arrayErr% = FN SETHANDLESIZE (recFieldPtr&.myArrayH&, newSize&)
      arrayPtr&.numElems% = newElemCount&
    END IF
  END IF
END FN = arrayErr%
```

requirements for more elements grow, so can the size of your array. If requirements decrease, the handle can be reduced to match.

The routine to resize an array handle can be seen in Program 89. The routine requires a pointer to the record field that contains the handle, and the new element count. The routine recalculates the appropriate size and resizes the handle if required, using the new value.

Cooldown

That finishes our tour of records. Along the way we learned what records are, how to create them, how to access their variables for writing and reading, and how to write them to disk and read them back later. We also saw how easy it is to pass record fields to an edit field for display and how to extract any new information from the edit field.

Finally, we saw how to handle resizeable arrays within a record structure by using handles and the `XREF@` statement.

CHAPTER 11

Files

Warm-up

This chapter introduces you to file handling on the Macintosh. In this chapter you will learn:

- How Macintosh files are organized,
- Two methods of locating files on Macintosh volumes,
- How to open, close, and get information about a file,
- Three ways of saving and reading file data, and
- How to use the standard open and save dialogs.

Macintosh Files

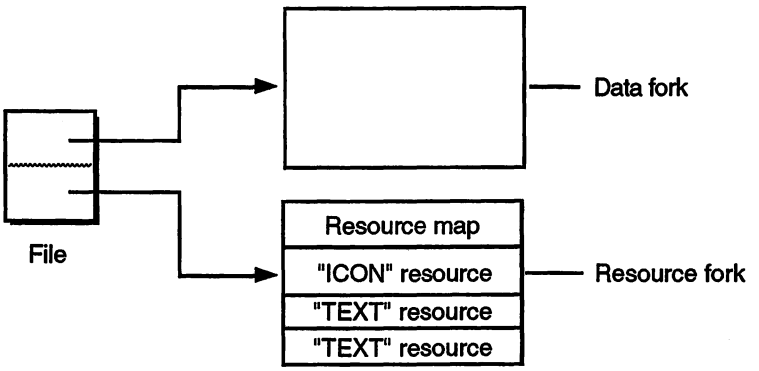
A **file** is data stored on disk. Files are created by an application as an ordered sequence of bytes on a Macintosh volume. A file can contain text, numerical data, images, and anything else a program can organize and write to disk.

Typically, files containing user data are referred to as documents when describing them to users. A **document** is any file a user can create or edit. A document has a specific file type. A **file type** is a 4-character alphanumeric sequence that describes the type of data the file contains (`TEXT`, `PICT`, or others). Some common file types can be opened by many applications (`TEXT`), while others can only be opened by the program that created them (`SbDb`, used by *SimpleBase* files).

Macintosh files contain two forks for storing information. One fork is called the data fork, the other is the resource fork. The **data fork** contains the file's data and is accessible using standard file commands (`OPEN`, `READ`, `WRITE`, `CLOSE`, and others). The **resource fork** contains file resources and is accessible using Toolbox commands. A file can contain a data fork, a resource fork, or both.

Resources are blocks of arbitrary amounts of data identified by a combination of name, resource type, and ID number. Some types of resources are common to many Macintosh applications and have a standardized format including `MENU`, `CODE`, `DLOG`, `ALRT`, `TEXT`, etc. For more information on resources, see the chapter "Resources".

FIGURE 48. Macintosh file forks.



Macintosh Volumes

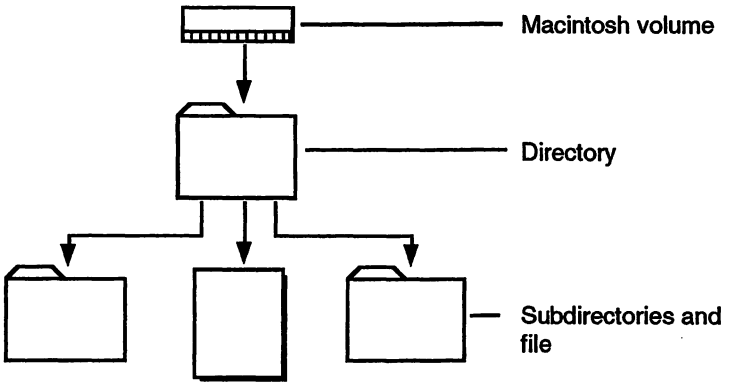
A **volume** is any storage device formatted to store files created by an application. Macintosh volumes are organized using folders. A **folder** is a subdivision of a Macintosh volume that can contain files or other folders. Folders are sometimes known as **directories**. Folders nested within other folders are also known as **subdirectories**.

Files can be located and accessed using a variety of means. We'll describe two common methods here: pathname and working directory reference number.

Full Pathnames

A **full pathname** is a series of concantenated folder names ending in a file name. A full pathname serves to uniquely locate a file by having the Macintosh File Manager walk a string of folder names until the file or folder is found. The folder name and the file name are separated from each other by a colon in a full pathname.

FIGURE 49. Macintosh volume design.



A full pathname might look like this:

```
MAC•1:Programming:Book Programs:15.Files(02)
```

Where "`MAC•1`" is the volume name and "`:Programming:Book Programs`" is the search path through two folders, and "`15.Files(02)`" is the file name. A full pathname tells a file opening routine how to locate the file by starting at a specific volume and leading it through the correct folders until the file is found.

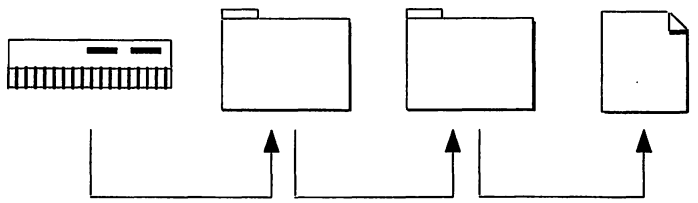
Using pathnames was somewhat common in the early years of Macintosh programming, but it does have a couple of problems on today's machines. The main problem is a pathname is stored as a string, so it's limited to only 255 characters. On today's larger hard drives folders can be buried many levels deep and a full pathname can easily exceed the length of a string. Even small drives can have trouble if a user assigns long descriptive folder names. Another problem is that any folder that has been renamed in the path will disrupt the file search totally. Figure 50 shows how a full pathname looks at every intermediate folder until it finds the specified filename.

So, using a full pathname has some advantages, but much better solutions are available as described in the next section.

Working Directories

A **working directory reference number** (WD) is a temporary reference number that combines a volume reference number with a directory ID to uniquely identify a folder. A working directory reference number is assigned by the operating system when a folder is opened and remains valid only while the folder is open. If the folder is closed and then reopened, it might have an entirely different value assigned to it.

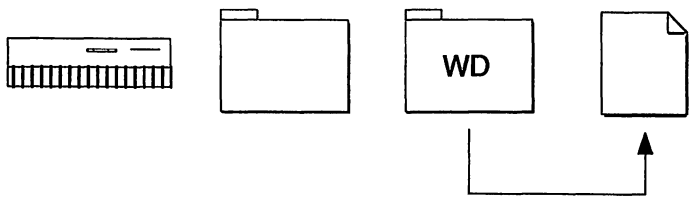
FIGURE 50. Searching by full pathname.



Using working directory reference numbers, the File Manager can go directly to a specified folder to locate a file. And since working directory reference numbers are easy to get with standard FB commands, they are simple and convenient to use.

For *SimpleBase*, we'll use the working directory reference number method of locating files. This is the one that the FB runtime supports and probably the one you should use in your own programs. Figure 51 shows the search path using just the filename and WD reference number. Note the lack of jumps between folders.

FIGURE 51. Searching by filename and working directory.



File Commands

FB contains several statements and functions that make creating and managing files quick and easy. Let's discuss each one before adding any file handling code to *SimpleBase*.

Opening a File

The standard statement for opening any file is `OPEN`. Opening a file called "Fred" for input looks like this:

```
OPEN "I", #fileID%, "Fred", 1, wdRefNum%
```

The OPEN statement requires five parameters. These parameters include the file permission, the file ID number, a filename, a record size, and a working directory reference number. The last two parameters, record size and WD reference number can be optional, although I strongly suggest you always use the WD number.

TABLE 8. File Privileges.

METHOD	ACCESS PERMISSION
I (input)	read-only
O (output)	write-only
A (append)	write-only
R (random)	exclusive read/write
N (network)	shared read/write

File Permissions

There are five methods described in Table 8 for opening a file. These methods determine how the OPEN statement in a program can interact with a file. Which method you use will be determined by what you are trying to accomplish with the file. For example, if you only need to read a text file you might use "I" (input). To write it back out to disk use the "O" (output) method. To open a file using write permission do this:

```
OPEN "O", #fileID%, "Fred", 1, wdRefNum%
```

And, to open a file with shared read/write permission use:

```
OPEN "N", #fileID%, "Fred", 1, wdRefNum%
```

Data vs. Resource Forks

Since there can be two forks associated with a file (data and resource), the OPEN statement also lets you open either file fork. In almost all cases, you will open the data fork. However, in rare circumstances you may need to open the resource fork. Once a file fork is open, this is where data will be read from disk using INPUT#, READ#, READ FILE# and written back to disk using PRINT#, WRITE#, or WRITE FILE#.

You specify which fork to open by appending a "D" (data) or an "R" (resource) to the access mode parameter. By default, if no fork is specified, the data fork is assumed. For example, to open the resource fork of a file do this:

```
OPEN "IR", #fileID%, "Fred", 1, wdRefNum%
```

We will only deal with a file's data fork in this chapter. While it's possible to access the resource fork using the OPEN statement I don't recommend it unless you have a very firm grasp of how a resource fork is organized. Fumbling around a file's resource fork is guaranteed to corrupt a file beyond redemption. In later chapters we'll describe how to access a file's resource fork safely using standard Resource Manager calls.

The fileID%

The next parameter OPEN expects is the `fileID%` (or `deviceID%`) itself. A `fileID%` is a positive integer limited by the maximum open files setting in the Preferences dialog. A `deviceID%` is normally negative and refers to a serial port or other device. We will only deal with `fileID%s` here.

Whenever FB opens a file, it allocates space for a file information buffer. A **file information buffer** is simply a block of memory reserved by FB for holding file information. This information includes the file size, type, creation and modification dates, as well as other information.

The number of files a program can have open at one time is limited to the number of file buffers allowed. Choose **Preferences** from the **Edit** menu to reset the maximum number of open files FB can handle at one time. The limit is 99. Increasing the open file limit increases the amount of space allocated for file buffers, so don't set it unnecessarily high.

Filenames

The `filename$` parameter is normally an individual file name, but can be a full or partial pathname containing volume and folder names, as well full a file name. See the "Full Pathnames" section for a complete description of pathnames.

Record Length

The `recordLength` parameter is used to specify the maximum size of an individual record contained within a file. A default length of 256 bytes is assigned by the runtime if a record length is not specified in the OPEN statement. You can safely leave this parameter blank if the data you are reading or writing doesn't have a default length.

Working Directory ID

The final parameter is the `wdRefNum%`. This is an integer value that uniquely identifies a folder in a particular volume. See the "Working Directories" section for additional details. It's the easiest method to use in FB and our preferred method for opening files in folders.

Getting the File Size

Once a file is open, it's possible to determine its size using the LOF function. Given a fileID%, LOF returns the size of the file fork opened with the OPEN statement. Here is an example of getting a file size (for the data fork):

```
OPEN "I", #fileID%, filename$, , wdRefNum%
fileSize& = LOF (fileID%, 1)
PRINT "The file "+filename$+" contains ";fileSize&;" bytes."
CLOSE #fileID%
```

The second parameter in LOF is used to specify a record length. When given a value of 1 (as shown above), it returns the byte count of the specified file fork. When given any other value, it returns a result that is the file size divided by that value. This comes in handy when dealing with records. For example, if we have a file that is 300 bytes in size and each record has a length of 60 bytes, LOF will return a count of five records.

To get the size of a file that contains both a data and a resource fork (as do *SimpleBase* files) use the subroutine shown in Program 90. It automatically opens and closes both file forks, getting the fork size each time, and returning the total size to the subroutine caller.

PROGRAM 90. Get real file size using OPEN.

```
LOCAL FN GetFileSize& (fileID%, filename$, wdRefNum%)
  OPEN "ID", #fileID%, filename$, , wdRefNum%
  dataSize& = LOF (fileID%, 1)
  CLOSE #fileID%
  OPEN "IR", #fileID%, filename$, , wdRefNum%
  rsrcSize& = LOF (fileID%, 1)
  CLOSE #fileID%
END FN = dataSize& + rsrcSize&
```

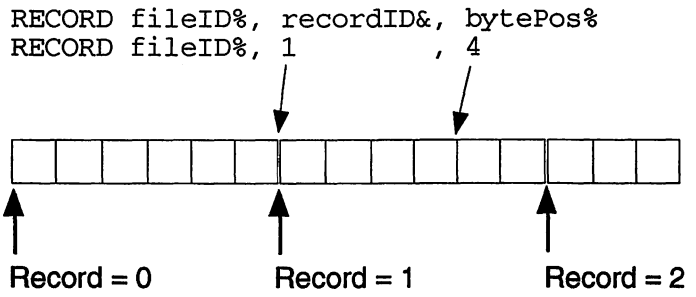
Setting File Positions

Once a file is open, we use the RECORD statement to position the file pointer within the file. A **file pointer** is the position in the file where the next read or write operation will start. RECORD is very flexible when it comes to setting the file pointer.

RECORD requires three parameters, the fileID%, a recordID&, and a bytePos%. The fileID% identifies which open file to operate on, the recordID& identifies the record position from the start of the file (if you're using records), and bytePos% specifies a byte offset from the start of the record. Thus, to specify byte 14 in record 125 in file 3 we can do this:

```
RECORD #3, 125, 14
```

FIGURE 52. Positioning the file pointer.



You can see how this works in Figure 52, when a file is opened which contains records 6 bytes long. We use RECORD to position the file pointer to the fourth byte of the first record.

- *Note that by default in FB, record zero is a valid record.*

Getting File Positions

Often, you will need to determine the file pointer position to know which record or byte offset in a record it is reading from. Use the REC and LOC functions to return this information.

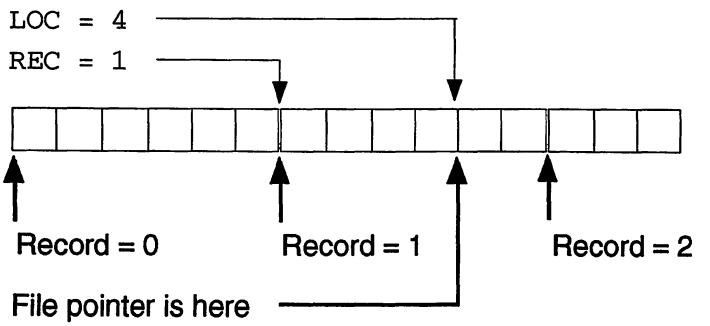
To determine the record number where the file pointer is currently positioned, use the REC function like this:

```
recordID% = REC(fileID%)
```

To get the byte offset within the current record, use the LOC function:

```
bytePos& = LOC (fileID%)
```

FIGURE 53. Record organization on disk.



Saving Data

Once a file has been opened, it's possible to save information to disk. FB offers you several methods of accomplishing this. The method you choose will depend upon the data to save.

PRINT#

When it comes to writing data to disk, the PRINT# statement is the statement of choice for most BASIC programmers. It allows them to write strings and numbers to disk without a whole lot of trouble. PRINT#, however, as an means of outputting data to disk is very slow. For example, to write a single string variable to disk you can do this:

```
PRINT #fileID%, tmp$
```

Or, you can write several numbers and strings to a file like this:

```
PRINT #fileID%, tmp$, myInt%, myLong&, mySp!
```

And, to print a dozen strings to disk as a TEXT file, use this:

```
DEF OPEN "TEXT????"
OPEN "O", #fileID%, fileName$, , wdRefNum%
FOR lineCount% = 1 to 12
  PRINT #fileID%, tmp$(lineCount%)
NEXT lineCount%
CLOSE #fileID%
```

This example opens the file for output only (since we're only sending data to disk), then loops through the dozen strings printing each to disk in turn. The DEF OPEN statement specifies the file type FB will assign to the file upon creation. We will talk more about file types later on in the "Getting the File Type" section.

The alter ego of PRINT# is the INPUT# statement. See the "INPUT# & LINE INPUT#" section for more details.

WRITE#

The WRITE# statement is a faster means of writing data to disk. WRITE# sends data to disk in the binary format used to store it in memory. The runtime writes the data to disk without any translation, reducing the calculation overhead and increasing output speed.

WRITE# already understands variable types like integers, long integers, single- and double-precision variables. It also understands record types. As with standard variables, the runtime already knows the record size and can treat it appropriately. Just give it the variable name of a record and let WRITE# do its job like this:

```
DIM myRecordVar.myRecordType
WRITE #fileID%, myRecordVar
```

The only variable type the runtime needs help with is strings. Since a string variable can range from 1 to 255 characters, WRITE# expects you to tell it how many characters to write to disk. This enables you to save space in a file by reducing the length of any string written to disk to the absolute minimum. To write a string variable simply do this:

```
author$ = "HEINLEIN"
WRITE #fileID%, author$;LEN (author$)
```

It's important to set the string variable accurately, because if you set the length too low, some characters will be lost. If you set it too high, WRITE# assigns additional space characters to fill up the length. To see results of either length error, examine Figure 54 which illustrates both problems.

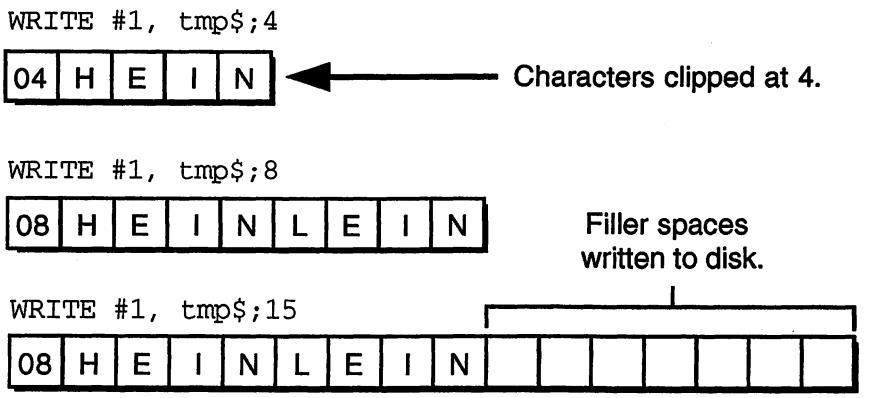
The alter ego of WRITE# is the READ# statement. See the "READ#" section for more details.

WRITE FILE#

For arrays, INDEX$, and other large data structures, neither of the previous two statements beat using WRITE FILE# in speed. Using the WRITE FILE# statement requires three parameters: a fileID%, a pointer to the data, and the number of bytes to write to disk. When executed, WRITE FILE# begins at the specified address, reads the specified number of bytes, then writes it to disk as a single block of data. Writing an entire 1M array to hard disk takes less than a second on a standard Macintosh II.

Program 91 demonstrates the speed of WRITE FILE# in place of traditional sequential methods, i.e., PRINT#. The Toolbox function TickCount returns the number of ticks between the start of the write operation and its ending

FIGURE 54. Writing strings to disk.



PROGRAM 91. Writing data to disk with WRITE FILE#.

```
_arrayElements = 100000
DIM myArray% (_arrayElements), endOfArray%
myArraySize& = @endOfArray% - @myArray% (0)
OPEN "O", #1, "WRITE FILE Test", , SYSTEM (_aplVol)
startTime& = FN TICKCOUNT
WRITE FILE#1, @myArray%(0), myArraySize&
stopTime& = FN TICKCOUNT
CLOSE #1
PRINT "Total ticks (1/60th Sec) =";stopTime& - startTime&
STOP
```

time. Each tick equals 1/60th of a second. Try increasing the `_arrayElements` constant beyond its current value to see how efficient `WRITE FILE#` really is.

Also, notice how we calculated the size of our array, by getting the address to the next `DIM`'ed variable minus the first element of the array. We did this using the shorthand version of `VARPTR` (the `@` sign). Since the runtime allocates memory to dimensioned variables in sequence, the `endOfArray%` variable comes right after the array in memory position, making it easy to calculate the array size.

Next, we open a new file and write the information in the array to disk. The Toolbox `TickCount` function gives us the start and stop times which we display at the end of the entire operation.

The alter ego of `WRITE FILE#` is the `READ FILE#` statement. See the section "READ FILE#" for more details.

Reading Data

Once the data is written to disk, getting it back into memory is not difficult. Mostly, it's a matter of reading the data back in using the same format as when it was written to disk.

INPUT# & LINE INPUT#

The `INPUT#` statement is the statement of choice to read data back into memory. Like `PRINT#`, it's a bit slow because of the string conversion overhead, but it's reliable and easy to use.

`INPUT#` is the mirror image of `PRINT#`. To read information from disk back into memory, copy the save disk routine and change all `PRINT#` statements to `INPUT#` statements. For example, to read a single string variable from disk:

```
INPUT #fileID%, tmp$
```

To read a combination of numbers and strings back into memory, use:

```
INPUT #fileID%, tmp$, myInt%, myLong&, mySp!
```

Also, LINE INPUT# works great if you know the file you're about to read is nothing but a collection of strings. To read a dozen strings previously saved as a TEXT file back into memory, copy the subroutine and change the PRINT# to LINE INPUT# like this:

```
OPEN "I", #fileID%, fileName$, , wdRefNum%
FOR lineCount% = 1 to 12
  LINE INPUT #fileID%, tmp$(lineCount%)
NEXT lineCount%
CLOSE #fileID%
```

READ#

READ# is the opposite of WRITE#, it accepts the same arguments used by WRITE# and reads the data back into the specified variables. As before, strings must be read using a length. And, as with PRINT# and INPUT#, create the read subroutine by copying the save subroutine and converting all WRITE# to READ# statements. For example, to read the same record used in the WRITE# example, do this:

```
READ #fileID%, myRecordVar
```

To read a string back, remember to assign the same length to the string variable used:

```
READ #fileID%, author$;15
PRINT author$
```

which will print:

```
HEINLEIN
```

Of course, you must be sure to specify the same number of characters to read as were previously written.

READ FILE#

READ FILE#, like WRITE FILE#, may be used to read large blocks of contiguous, arbitrary blocks of data from disk into memory. READ FILE# requires a fileID%, a pointer to where to place the data, and a byte count.

It's important to remember that when reading disk data into memory, ensure that you have enough space set aside to accept the data and avoid trampling over other variables already stored in memory. A mistake here can cause all kinds of problems.

PROGRAM 92. Reading data from disk with READ FILE#.

```
_arrayElements = 100000
DIM myArray% (_arrayElements)
OPEN "I", #1, "READ FILE Test", , SYSTEM (_aplVol)
fileSize& = LOF (1,1)
startTime& = FN TICKCOUNT
READ FILE#1, @myArray%(0), fileSize&
stopTime& = FN TICKCOUNT
CLOSE #1
PRINT "Total ticks (1/60th Sec) =";stopTime& - startTime&
STOP
```

Setting a File Type

Every file on a Macintosh has a file type associated with it. Some file types are quite common, like TEXT or PICT, and can be opened by many applications. Others have unique file types that are opened only by the application that created it. To set a file type use the DEF OPEN statement.

DEF OPEN specifies both the file type and application signature of the program that created the file. Once defined, it remains in effect until another DEF OPEN statement is encountered in the program.

The most common way to make sure a program file has the required file type is to define the type prior to opening the file. In *SimpleBase*, we do it this way:

```
DEF OPEN "SbDbFbSb"
OPEN "R", #1, filename$, , wdRefNum%
```

This ensures that any file we save always has the correct file type associated with it.

Closing a File

Once you're finished with a file, it's always a good idea to close it until it's needed again. The more files a program has open, the more memory required, and the greater possibility of data loss or file corruption should a power outage or system error occur. The best means of combating both of these problems is to open the file, read the data required into memory, then close it immediately.

A convenient way to close a file or other open device is to use the CLOSE statement like this:

```
CLOSE #fileID%
```

With use of the optional `fileID%`, it's possible to close any file, while leaving others untouched. However, you can close all currently open files, devices, or ports by using `CLOSE` without any parameter like this:

```
CLOSE
```

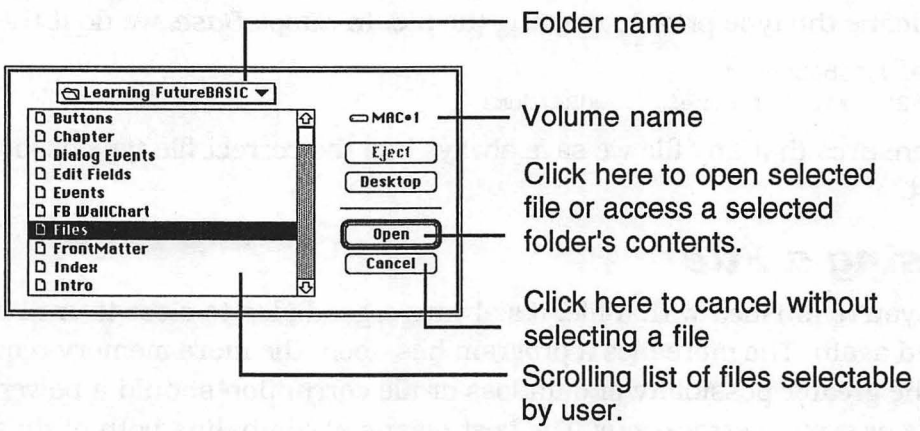
The `RESET` keyword provides the same functionality of `CLOSE` without the parameters. Use it exactly as `CLOSE` to close all files or devices opened by the program.

Open File Dialog

The Macintosh has always provided an uniform method of accessing a file on disk. This method is known as the standard get file dialog. FB enables you to access this dialog using the `FILES$ _fOpen` function. See Figure 55 for an illustration of a standard get file dialog.

`FILES$` accepts three parameters, the `_fOpen` constant, a file type string, and a working directory reference number. The additional parameter positioned between file types and WD number is ignored when using the `_fOpen` version of `FILES$`. Unlike most other functions, `FILES$ _fOpen` returns two pieces of information, the filename and the file's working directory reference number.

FIGURE 55. Standard get file dialog.



The scrolling list that appears in the dialog will normally display all the files in the current directory unless we limit those choices. `FILES$` can filter the filenames that appear in its scrolling list by file type (`TEXT`, `PICT`, etc.). Up to four file types can be filtered at one time. This means users don't have to view every file when looking to open only program files, instead, they will only see files the program can open. For example, to see only `TEXT` type files in the dialog use this:

```
filename$ = FILES$ (_fOpen, "TEXT", , wdRefNum%)
```

To see two file types, concantenate the two types into a single string variable like this:

```
filename$ = FILES$ (_fOpen, "TEXTPICT", , wdRefNum%)
```

Now the dialog will show only files of types TEXT and PICT.

Just as with anything else, check to ensure you have a valid filename before attempting to open the file. A good way to do this is to use the LEN function. If the user clicks the **Cancel** button in the FILES$ dialog, the filename will be blank and its length set to zero.

We can test condition like this:

```
filename$ = FILES$ (_fOpen, "TEXT", , wdRefNum%)
LONG IF LEN(filename$) > 0
  'do something with filename$
END IF
```

Getting the File Type

Since it's possible to display many different file types in a FILES$ _fOpen dialog, we need a method of determining the file type in order to properly open the file. We can determine a file's type using an alternate form of FILES$, one without parameters.

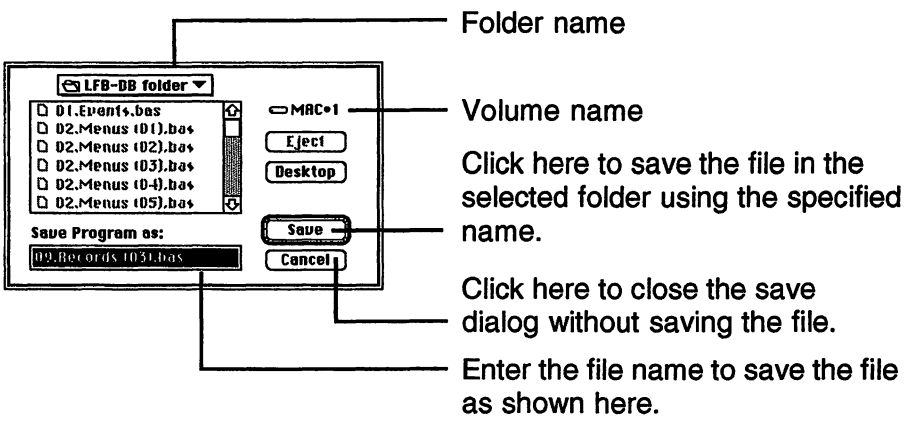
Look at our last example that had a dialog that displayed both TEXT and PICT file types. The method of reading data from each type will definitely be different, so we need to know which type of file the user chose so that the correct open subroutine is called. Here is one way that would read the file type and branch to the correct opening routine:

```
filename$ = FILES$ (_fOpen, "TEXTPICT", "", wdRefNum%)
LONG IF LEN (filename$) > 0
  SELECT FILES$
    CASE "PICT" : FN OpenPICTFile (filename$, wdRefNum%)
    CASE "TEXT" : FN OpenTEXTFile (filename$, wdRefNum%)
  END SELECT"
END IF
```

Save File Dialog

Another valuable Macintosh feature is the save file dialog. This dialog normally appears to request a filename and folder when saving a file. Unlike the FILES$ _fOpen function, FILES$ _fSave requires all four parameters, the _fSave constant, the message string, the default filename that appears in the dialog, and a wdRefNum%. Figure 56 shows a typical save file dialog.

FIGURE 56. Standard save file dialog.



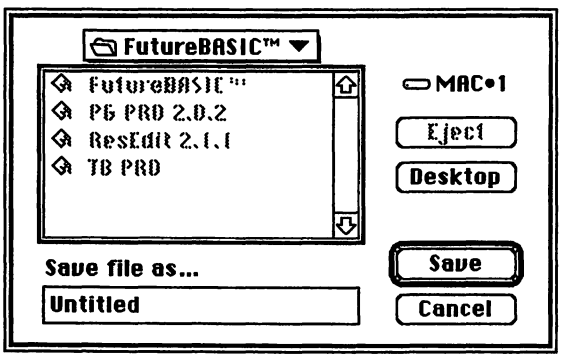
To specify a default filename and the save message that appears in the dialog, you might do something like this:

```
filename$ = FILES$(_fSave, "Save program as:", "Untitled", wdRefNum%)
```

And it would appear as shown in Figure 57.

In addition to returning the filename the user enters in the text field of the dialog, `FILES$ _fSave` also returns the `wdRefNum%` to the chosen folder.

FIGURE 57. A modified save dialog.



Handling Folders

The `FILES$` functions provide a familiar method for a user to navigate the volumes and folders available to them. The following functions enable your program to navigate them just as easily as a user.

Finding Folders

Many times you will need to determine the folder where the program resides. You can do this with the SYSTEM function. To retrieve the WD reference number of the folder where the program is located use:

```
currentPgmWDRefNum% = SYSTEM (_aplVol)
```

You can locate files stored in the same folder as the program. This location is often used by a program to store preference files. Another popular location for storing preference files under System 6 is the System folder. To locate the System folder use:

```
systemWDRefNum% = SYSTEM (_sysVol)
```

Under System 7, preference files should be stored in the Preferences folder within the System folder. You can store files there using a combination of SYSTEM (_sysVol) and a partial pathname. The search starts in the System folder, then walks the pathname until the file is found. For example, to save a file in the Preferences folder do this:

```
pathname$ = ":Preferences:My Prefs File"
OPEN "O", #1, pathname$, , SYSTEM (_sysVol)
```

You can also retrieve the WD reference number of the Preferences folder using the FOLDER function. To do that use these lines:

```
sysWDRefNum% = SYSTEM (_sysVol)
sysWDRefNum% = FOLDER ("", sysWDRefNum%)
prefWDRefNum% = FOLDER ("Preferences", 0)
```

We start by getting the System folder's WD number, switch the active directory to it, then see if a Preferences folder exists. If the Preferences folder exists, we get a valid WD number. If not, a zero is returned. Use this same technique to determine if any folder already exists in the current folder. Start by getting the current active folder:

```
currentWDRefNum% = FOLDER ("", 0)
```

And follow it with this:

```
foundWDRefNum% = FOLDER ("This Folder", 0)
```

Where "This Folder" represents the name of the folder to locate. If the folder exists, foundWDRefNum% will return its WD reference number. If the folder doesn't exist, zero is returned.

Creating Folders

You can also create your own folders using the FOLDER function. FOLDER requires two parameters, the name of the folder to create and a valid WD reference ID where to place the folder. For example, the following line will

create a new folder called "Program Stuff" in the same directory as the program itself:

```
fWDRefNum% = FOLDER ("Program Stuff", SYSTEM (_aplVol))
```

The program's OPEN statements can then use the fWDRefNum% variable to read and write files to the new folder.

Regular Exercises

Now that we know how to handle files on the Macintosh, let's begin implementing them in *SimpleBase*.

Saving a File

The first thing to do is save some employee data to disk so that it can be read later. The pseudocode to accomplish this operation looks like this:

```
1. Open the file
2. Set the record position to write
3. Write record data
4. Update maximum record count
5. Close file
```

The subroutine to handle this activity is called FN DBWriteRecord. It starts by opening the file in "R" mode and setting the file position with RECORD. Once the file is open, it writes the data in the gEmployee record using WRITE# at the specified record number, updates the gMaxRecInFiles% variable using the LOF function, and finally closes the employee file.

PROGRAM 93. DBWriteRecord subroutine.

```
LOCAL FN DBWriteRecord
  DEF OPEN "SbDbFbSb"
  OPEN "R", #gFileNum%, gFileName$, _dbRecordSize, gWdRefNum%
  RECORD #gFileNum%, gRecordNum%, 0
  WRITE #gFileNum%, gEmployee
  gMaxrecinFiles% = LOF (gFileNum%, _dbRecordSize)
  CLOSE #gFileNum%
END FN
```

Opening a File

After we've saved some employee data to disk, we read it back into memory by reversing the process used to save it. The same general pseudocode describes the entire read file operation:

```
1. Open the file
2. Set the record position to read
```

```
3. Read record data
4. Update maximum record count
5. Close file
```

As soon as the file is open, FN DBReadRecord uses READ# to read the record data from disk into the global record variable gEmployee. Next, it updates the global variable gMaxRecInFile%, then closes the file.

PROGRAM 94. DBReadRecord subroutine.

```
LOCAL FN DBReadRecord
  DEF OPEN "SbDbFbSb"
  OPEN "R", #gFileNum%, gFileName$, _dbRecordSize, gWdRefNum%
  RECORD #gFileNum%, gRecordNum%, 0
  READ #gFileNum%, gEmployee
  gMaxRecInFile% = LOF (gFileNum%, _dbRecordSize) - 1
  CLOSE #gFileNum%
END FN
```

File Handling

Our file handling subroutines are now in position. We still need to add calls to them in other subroutines so that *SimpleBase* can become a real working program. Let's begin with the routines to create a new employee file.

New Employee Files

Creating a new employee file requires some setup before it's displayed to the user. The pseudocode to accomplish this whole operation is as follows:

```
1. Get a name for the employee file
2. Clear old data from employee record
3. Create a new employee file
4. Set empty record data
5. Write blank record to file
6. Close file
7. Build data entry window
8. Update window edit fields
```

The responsibility for creating a new file falls to the FN ItemNew subroutine. The entire code to implement a new file is shown in Program 95. It starts by getting a filename and WD reference number using FILES$. If it gets a valid name, it clears any data from the employee record, resets the gOpenRecord% variable to zero and calls FN DBNewDataBase.

The subroutine DBNewDataBase shown in Program 96 starts by assigning some default information strings to the first few fields of the employee record. These will be written to disk in record number zero to help identify the creator of the file. Again, this is not required but does make it nice if you ever need to

PROGRAM 95. ItemNew subroutine.

```
LOCAL FN ItemNew
  gFileName$ = FILES$ (_fSave, "Save database as:", "Untitled", gWdRefNum%)
  LONG IF LEN (gFileName$) > 0
    DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
    gOpenRecord% = 0
    FN DBNewDataBase
    DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
    gEmployee.dbName$ = "Empty record"
    gEmployee.dbDeptNum% = _programBTN
    FN DBWriteRecord
    FN WindowBuild (_dbEntryWIND)
    FN EFRecordToEF
  END IF
END FN
```

recover the file from a crashed disk. Next, the employee record containing the info strings is written to disk with `DBWriteRecord`, and the `gOpenRecord%` variable is incremented to point to the next record.

Upon return to `FN ItemNew`, the info strings are again cleared from the employee record, new default information is assigned so that we know it's a new record, and the first real employee record is written to disk. Once safely on disk, the Data Entry window is built using `FN WindowBuild`, and the default employee data is sent to the window's edit fields with `FN FieldRecordToEF`.

PROGRAM 96. DBNewDataBase subroutine.

```
LOCAL FN NewDatabase
  gEmployee.dbName$ = "This file was created by SimpleBase"
  gEmployee.dbAddr$ = "from the book: Learning FutureBASIC."
  gEmployee.dbCity$ = "Published by Sentient Fruit™"
  FN DBWriteRecord
  INC (gRecordNum%)
END FN
```

Opening Employee Files

Of course, the most natural place to open an employee file is with **Open** on the **File** menu. Therefore, rewrite the `FN ItemOpen` to look like the one in Program 98.

It begins with `FILES$` to get a filename and a WD reference number. If `gFileName$` is valid, we use a `DEF BLOCKFILL` to erase the current information

in the gEmployee record. This isn't mandatory but I like to start with a clean record slate anyway.

Next, the subroutine sets a couple of global variables, then calls the FN WindowBuild subroutine to construct the Data Entry window. When the window is built, it calls DBReadRecord to get the first record from the file. Finally, it transfers the data in the gEmployee record to the data entry window's edit field using FN EFRecordToEF.

Note that none of this happens unless the user selects a valid employee file from disk. We limit which files are shown in the FILES$ dialog by using the file filter type SbDb.

PROGRAM 97. Revised ItemOpen subroutine.

```
LOCAL FN ItemOpen
  gFileName$ = FILES$ (_fOpen, "SbDb", , gWdRefNum%)
  LONG IF LEN (gFileName$) > 0
    DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
    gRecordNum% = 1
    FN WindowBuild (_dbEntryWIND)
    FN DBReadRecord
    FN EFRecordToEF
  END IF
END FN
```

The next place we need to read an employee record is in the FN DoRecordMenu subroutine just before END FN and after the END SELECT statement. By placing it here, we enable the menu items associated with maneuvering the records. Commands like First, Last, Previous, and Next will now work. The lines of code to do this are:

```
LONG IF itemID% < _iClearRec
  FN DBReadRecord
  FN EFRecordToEF
END IF
```

This sequence is repeated in several places throughout the listing. It is called in the DialogGotoWindow, DBFindRecord, DialogEntryWindow, and PrintManyRecords subroutines. Examine the complete *SimpleBase* program in the Appendix for details.

Saving Employee Files

As with DBReadRecord, DBWriteRecord is called numerous times throughout the program. The obvious place to look is the FN ItemSave subroutine called

when **Save** is chosen from the **File** menu. The complete `ItemSave` subroutine is shown in Program 98.

`FN ItemSave` first determines if it has a valid filename. It clears the old data from the `gEmployee` record with `DEF BLOCKFILL`, then calls `FN EFToRecordField` to return the latest version of the employee data from the active Data Entry window. Finally, it calls `DBWriteRecord` to open the specified employee file and save the new information.

PROGRAM 98. Revised ItemSave subroutine.

```
LOCAL FN ItemSave
  LONG IF LEN (gFileName$) > 0
    DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
    FN EFToRecord
    FN DBWriteRecord
  END IF
END FN
```

As before, the other major subroutine that uses `DBWriteRecord` is `DoRecordMenu`. Because we save the latest information each time the user selects first, last, previous, or next record, either by button or menu, `DBWriteRecord` gets called before reading the next record from disk. Enter the following lines before the `SELECT itemID%` statement in `DoRecordMenu`:

```
LONG IF itemID% < _iClearRec
  FN EFToRecord
  FN DBWriteRecord
END IF
```

`DBWriteRecord` is also called from the following subroutines: `WindowCapture`, `DBNewDataBase`, `ItemNew`, `ItemClearRecord`, and `DialogEntryWindow`. Examine the complete *SimpleBase* program listing in the Appendix for more details.

Record Creation

A couple of routines still need to be expanded or introduced. First, we need some method of adding new records to the employee file. We also need some way of changing the value of `gOpenRecord%` so that the menu and button choices to move in the file actually work.

To add the same response to the **New** item of the **File** menu requires some window trickery. We have already written it so that it creates a new employee file, how can we change it to also create new employee records? The answer is simple, we know that **New** should only create employee files *if the Data Entry*

PROGRAM 99. Revised ItemNew subroutine.

```
LOCAL FN ItemNew
  LONG IF WINDOW (_outputWClass) = _dbEntryWIND
    FN EFToEFRecord
    FN DBWriteRecord
    gRecNumber% = gMaxRecords%
    DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
    FN DBWriteRecord
    FN EFRecordToEF
  XELSE
    gFileName$ = FILES$ (_fSave, "Save database as:", "Untitled", gWdRefNum%)
    LONG IF LEN (gFileName$) > 0
      DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
      gFileNum% = 1
      gRecordNum% = 0
      FN NewDatabase
      DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)
      gEmployee.dbName$ = "Empty record"
      gEmployee.dbDeptNum% = _programBTN
      FN DBWriteRecord
      FN WindowBuild (_dbEntryWIND)
      FN EFRecordToEF
  END IF
END FN
```

window is absent. But, if it's present on the screen, we should instead create a new record in the file. The routine to handle this is shown in Program 95.

Once this code is in place, its easy to implement from the **New Record** button. Find the `FN DialogEntryWindow` subroutine and add a call to the `FN ItemNew` subroutine in response to a button click in **New Record**. Just add the line inside the `SELECT dlgID%` structure like this:

```
SELECT dlgEvnt%
  CASE _btnClick
    SELECT dlgID%
      CASE _newRecordBTN
        FN ItemNew <<-- ADD THIS LINE
    END SELECT
END SELECT
```

Navigating Records

Finally, we need to add the means to set the `gOpenRecord%` variable.

Records Menu

The **Records** menu handles the majority of our gOpenRecord% manipulation. It is here that we can quickly pick the first, last, previous, or next record number via menu or button.

One important difference in the DoRecordMenu subroutine shown in Program 100 has to do with record saving. We add calls to our record handling subroutines to ensure that each record is saved before we move to another file. For example, if the user chooses **Previous**, DoRecordMenu first calls EFtoRecordField and DBWriteRecord to store the current record's data. It then calls the ItemPrevRecord function to change gOpenRecord%, and finishes by reading the specified record from disk with DBReadRecord and showing it with RecordFieldToEF. Because of this, the user never has to worry about saving data, it's all handled automatically.

PROGRAM 100. Revised DoRecordMenu subroutine.

```
LOCAL FN DoRecordMenu (itemID%)
  FN EFtoRecordField
  FN DBWriteRecord
  SELECT itemID%
    CASE _iFirstRec      : FN ItemFirstRecord
    CASE _iPrevRec       : FN ItemPrevRecord
    CASE _iNextRec       : FN ItemNextRecord
    CASE _iLastRec       : FN ItemLastRecord
    CASE _iFindRec       : FN ItemFindRecord
    CASE _iGotoRec       : FN ItemGotoRecord
    CASE _iClearRec      : FN ItemClearRecord
  END SELECT
  FN DBReadRecord
  FN RecordFieldToEF
END FN
```

The subroutines to handle the various manipulations of gOpenRecord% are shown in Program 101. As you can see, they all set gOpenRecord%, but in the subroutine ItemPrevRecord and ItemNextRecord it's very important to make sure we don't exceed the valid record boundaries and to avoid any file errors.

Find Record

Choosing **Find...** from the **Records** menu or using the **Find** button in the Data Entry window enables us to search for a record in our database. As it's designed now, it will only search the dbName$ field of the employee record. Let's modify the DialogFindWindow button event's section so that it looks like that shown in Program 102.

PROGRAM 101. Moving through the employee file.

```
LOCAL FN ItemFirstRecord
  gOpenRecord% = 1
END FN
LOCAL FN ItemPrevRecord
  DEC (gOpenRecord%)
  IF gOpenRecord% < 1 THEN gOpenRecord% = gMaxRecInFile%
END FN
LOCAL FN ItemNextRecord
  INC (gOpenRecord%)
  IF gOpenRecord% > gMaxRecInFile% THEN gOpenRecord% = 1
END FN
LOCAL FN ItemLastRecord
  gOpenRecord% = gMaxRecInFile%
END FN
```

As you can see, we call `WindowClose` twice, which in turn calls `WindowCapture`, so let's look at it in Program 103 to see how it works. It extracts whatever the user enters in the Find window's edit field into a new global variable called `gSearch$`. Then, when control returns to `FNWindowClose`, the Find window itself is closed. Finally, depending upon which button was chosen, we clear the `gSearch$` for a cancel or call the `DBFindRecord` function to begin the search.

PROGRAM 102. Revised DialogFindWindow.

```
LOCAL FN DialogFindWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _findBTN
          FN WindowClose (_dbFindWIND)
          FN DBFindRecord
        CASE _cancelBTN
          FN WindowClose (_dbFindWIND)
          gSearch$ = ""
        CASE _ignoreCaseBTN
          gCaseFlag% = FN CheckBoxHandler (dlgID%)
      END SELECT
    CASE ELSE
  END SELECT
END FN
```

PROGRAM 103. WindowCapture's Find window section.

```
LOCAL FN WindowCapture (wndID%)
  closeFlag% = _true
  SELECT wndID%
    CASE _dbFindWIND
      gSearch$ = EDIT$(_dbFindFLD)
  END SELECT
END FN = closeFlag%
```

DBFindRecord is a simple, sequential search routine shown in Program 104. It starts by storing the current gOpenRecord% value, and converts gSearch$ to all caps if that option was chosen in the Find window. Then, beginning with record number one, it cycles through each record in turn, using DBRead-Record to search for a match using the INSTR function. If a match is made, the found flag causes the routine to exit the loop, and the current record is shown in the Data Entry window. If no match is found, the loop exits when there are no more records to read, the original record number is restored and read back into memory. Later, we'll see how to add an alert to tell the user that no match was found.

PROGRAM 104. Finding a specific record.

```
CLEAR LOCAL
LOCAL FN DBFindRecord
  originalRecNum% = gOpenRecord%
  IF gCaseFlag = _markedBtn THEN gSearch$ = UCASE$ (gSearch$)
  CURSOR _watchCursor
  gOpenRecord% = 1
  DO
    FN DBReadRecord
    test$ = gEmployee.dbName$
    IF gCaseFlag = _markedBtn THEN test$ = UCASE$ (test$)
    found% = INSTR (1, test$, gSearch$)
    INC (gOpenRecord%)
  UNTIL (found% <> 0) OR (gOpenRecord% > gMaxRecInFile%)
  CURSOR _arrowCursor
  LONG IF found% = 0
    BEEP
    gOpenRecord% = originalRecNum%
    FN DBReadRecord
  END IF
  WINDOW #_dbEntryWIND
  FN RecordFieldToEF
END FN
```

Goto Record

Choosing **Goto...** from the **Records** menu allow us to specify which record to view directly, without flipping through the dozens of records between where we are and where we want to be.

PROGRAM 105. Capturing the new record number.

```
LOCAL FN WindowCapture (wndID%)
  SELECT wndID%
    CASE _gotoWIND
      gOpenRecord% = VAL (EDIT$(_gotoFLD))
      IF gOpenRecord% < 1 THEN gOpenRecord% = 1
      IF gOpenRecord%>gMaxRecInFile% THEN gOpenRecord% = gMaxRecInFile%
  END SELECT
END FN
```

Going to a specific record starts in the `DialogGotoWindow` function where we save the current record and store a copy of the current record number. Then extract a new record number from the Goto window's edit field using `FN WindowCapture` as shown in Program 106. In `WindowCapture`, it's converted from a string into the `gOpenRecord%` number, then checked to make sure it fits the current file number boundaries. Lastly, `FN WindowClose` closes the Goto window.

PROGRAM 106. Going to a specific record.

```
LOCAL FN DialogGotoWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    CASE _btnClick
      SELECT dlgID%
        CASE _gotoBTN
          FN DBWriteRecord
          FN EFtoRecordField
          originalRecNum% = gOpenRecord%
          FN WindowClose (_gotoWIND)
          LONG IF dlgID% = _gotoBTN
            FN DBReadRecord
            FN RecordFieldToEF
          XELSE
            gOpenRecord% = originalRecNum%
          END IF
      END SELECT
    CASE ELSE
  END SELECT
END FN
```

When control returns to DialogGotoWindow, it looks at which button was selected. If cancelled, the code just restores the original record number, otherwise, it reads in the specified record and displays it. The entire subroutine to handle this is shown in Program 106.

Our file handling is complete. You should now try out *SimpleBase* by creating some employee files. Save a couple and try re-opening them.

Peak Performance

To round out your knowledge of file handling here is a convenient method of determining if a file exists in the chosen folder.

Is File There?

We finish with a small subroutine which allows you to determine if a file exists in a specified folder. It makes use of the Toolbox function GetFileInfo to see if the file is there. If the routine in Program 106 returns anything but a zero, the file is missing from the designated folder.

PROGRAM 107. Does the file exist?

```
CLEAR LOCAL
DIM pbBlk.80
LOCAL FN FileExists% (fileName$, wdRefNum%)
  pbBlk.ioVolName& = @fileName$
  pbBlk.ioVRefNum% = wdRefNum%
  fileMissing% = FN GETFILEINFO (@pbBlk)
END FN = fileMissing%
```

Cooldown

That wraps it up for files and file handling. Along the way we talked about the various routines like OPEN, CLOSE, RECORD, POS, LOC, REC, and many more that enabled you to not only open disk files, but gather information about them. We also saw how to implement the standard open and save dialogs using FILES$ and how to get a file's type. Finally, we saw how easy it is to add a few reading and writing routines to *SimpleBase* and make it a real working program.

CHAPTER 12

Globals & Includes

Warm-up

Up to now, we've worked with the *SimpleBase* program as a single file. For small projects this may be an ideal solution. But for larger projects, having all your code in a single file may not make sense. In this chapter we will:

- Learn what global and include files are,
- Learn the benefits of using global files,
- Learn the benefits of using include files, and
- Learn how to use global and include files.

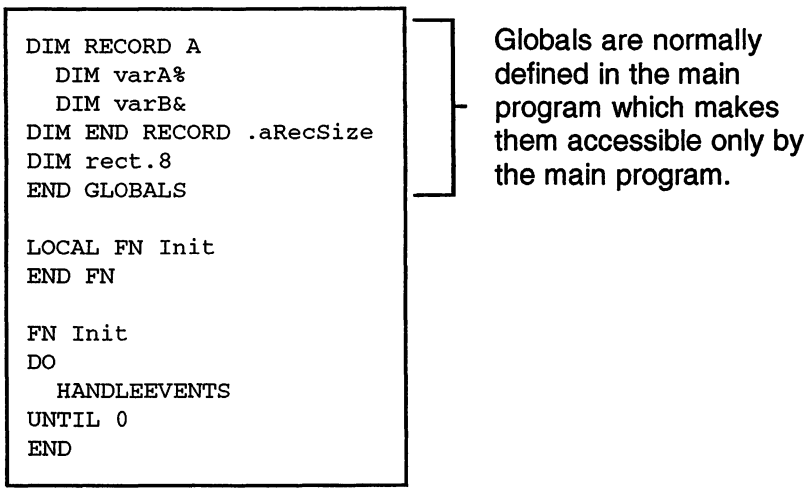
What are Globals?

A **global file** is a document that contains the definitions of all program records, dimensioned variables, and arrays. Normally a program uses a single global file, but can use several if required.

The most important reason to move global definitions from the main program file to a separate global file, is to allow other program files (called includes – which we'll look at later), easy access to the same information without repeating the globals in every file.

Imagine a global file to be a sheet of music. If the members of an orchestra were each given a different music sheet, it's unlikely that they could carry a tune when they attempted to play together. Yet if each has the same music sheet, wonderful music is usually the result. It's the same when writing a program. If each program file (both main and include) uses different sheets of

FIGURE 58. Regular use of globals variables.



global music, it's doubtful the program will execute successfully. Yet, if each program file executes while using the same global file, a working program is the result.

Examine Figure 58 to see a program with global variables. Here the defined variables can be read only by the main program. In Figure 59, we can look at a program that has a separate globals file. As a separate file, the globals defined there are accessible not only to the main program, but other program files too. This ability to separate files will become even more important the larger your programs become. Instead of redefining your global variables in each file, you can make a single change in the globals file, and know that all of the program files will see the change.

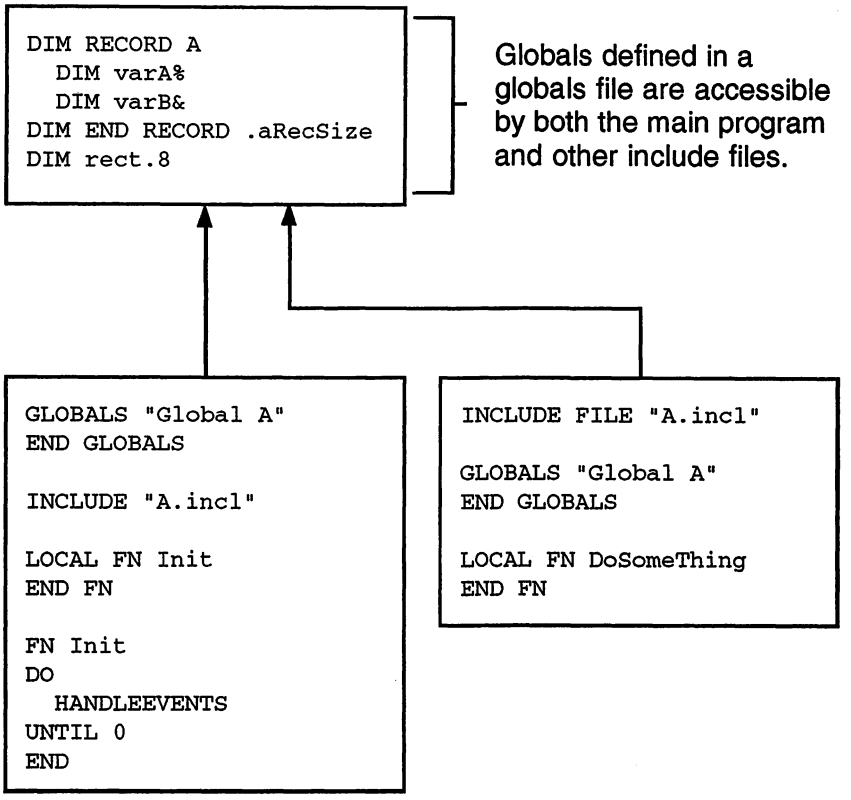
There's one point to remember. The definitions used in a global file are available to all portions of a program. Make sure that the variables and records you define there are required by the entire program before assigning them to the global file.

Creating a Global File

It's not difficult to create a globals file. Just create a new document in FB and begin entering constant and variable definitions, record structures, and arrays. Then, save it as a `TEXT` file (the tokenized file format is not allowed) using the suffix "`.glbl`" (case is unimportant on suffixes). This helps to identify the file properly in the Project tool's window.

Note, while you can dimension variables and arrays in the global file, don't try to add data to them. The runtime uses a small subset of routines to read and

FIGURE 59. Globally available globals.



assimilate the definitions in the global file. Attempts to set global variables may not work at all. Set any array values in your program's initialization routines during program start-up.

Accessing a Global File

Once your project has a global file, you make it available to any program file (main or include) using the GLOBALS statement like this:

```
GLOBALS "SimpleBase.glbl"
END GLOBALS
```

Always follow the GLOBALS statement by an END GLOBALS statement to identify the end of global declarations in the program file. This procedure should be followed in every file of a project that requires access to the global definitions.

Global File Do's & Don't's

Here are some general rules of advice to follow when using global files in your projects:

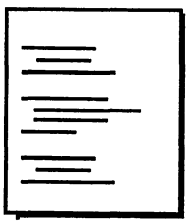
- *Do* include all constant definitions, as well as globally defined `DIM` variables, arrays, and record structures.
- *Don't* initialize any global variables in the global file. Wait to do that in your program's initialization routines.
- *Do* try to keep global declarations to a minimum.
- *Don't* forget to update include files when a new global variable, array, or record is added or removed from the globals file.
- *Do* identify global files with the suffix "`.glbl`" so that the Project Manager tool can display the file properly.
- *Do* remember to identify your global variables as global. The standard we suggest is the lowercase "g" prefix on variable names. If you use something different, make it consistent over all of your programs.

- *A small caveat, an include file that has already been compiled must be re-compiled before it will see any changes in the globals file. See the Includes section for more details.*

What are Include Files?

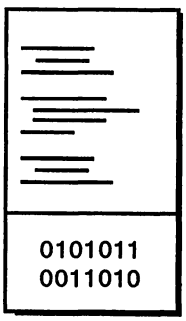
An **include file** is a program file that contains both source code and compiled code. Remember, files can contain both data and resource forks, and an include file has both. The data fork holds the source code and its compiled code is stored in the resource fork. This is shown graphically in Figure 60.

FIGURE 60. Include file.



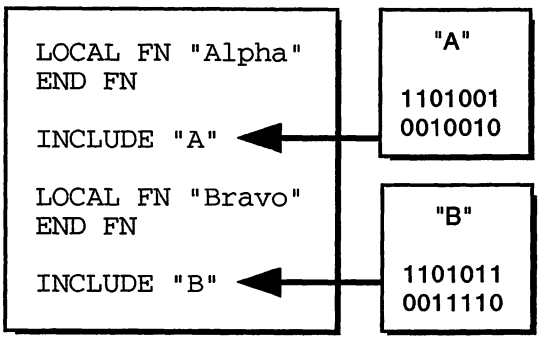
An uncompiled include file contains only source code just like the main file.

A compiled include file contains both source code and one or more code resources containing compiled code.



One reason to use an include file is speed. For example, when it comes time to build or run your program, the compiler proceeds to compile source code into machine code. When it encounters an INCLUDE statement in a file (whether main or include), it opens the designated include file and replaces the single INCLUDE statement with all of the compiled code in the include file, then continues compiling the rest of the main program's source code. This insertion of already compiled code greatly speeds up the compilation process. There is no limit to how many include files that can be used in a single program.

FIGURE 61. Inserting includes into a main program.



When the compiler encounters an INCLUDE statement, it replaces it with the pre-compiled code found in the specified include file.

Another reason to use include files is re-usability. It's possible to write a routine once, and then re-use it again and again in other programs.

Include File Types

There are three types of include files that FB can create. Each are identified by a constant value that makes it easy to remember which one does what. These constants and descriptions can be seen in Table 9.

TABLE 9. Include file types.

_Constant	Description
_aplIncl	Creates an include file using the entire runtime package. All FB and Toolbox commands are available.
_resIncl	Creates an include file that uses only the mini-runtime package. That means you are restricted to functions and procedures in the Toolbox as well as commands in the *Reference* manual marked with the Ⓜ.
_allIncl	Creates an include file using both the mini-runtime and the full runtime package.

- *Note that use of the `_resIncl` and `_allIncl` restricts source code statements in the include file to Toolbox routines and those few BASIC statements understood by the mini-runtime.*

Include File Limitations

While include files offer a host of utility to the programmer, there are some restrictions that do apply.

The 32K Limit

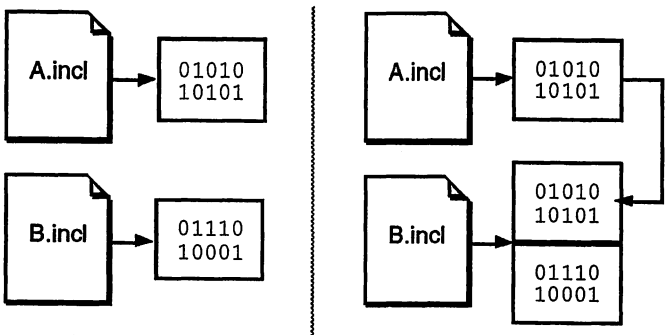
Include files are limited to 32K of compiled code. Due to the fundamental design of Macintosh memory, a single compiled code segment is limited to no more than 32K. That means, when it comes time to compile an include file, all of the compiled code must remain within this 32K limit. If the code segment exceeds this, you will get the following error:

```
Code segment too large for compiling.
```

Another problem manifests itself when an attempt is made to combine two include files together. In this case, the total compiled code for both can't be larger than 32K. This is shown graphically in Figure 62. The individual include files on the left each have 32K limits. However, on the right, B.Incl now has used the `INCLUDE` statement to add all of A.Incl's compiled code to itself. This compiled combination must also stay within the segment size limitation. If the total of both exceed 32K, you'll get an error.

A solution to this problem is to examine both files and remove, or move to another include, any redundant or unnecessary routines until the total is less than 32K in size. It's also possible to redirect program control using a technique described later in the Peak Performance section.

FIGURE 62. Combining Include file limits.



If A.incl and B.incl do not call each other, each has a 32K limit.

However, if B.incl INCLUDEs A.incl. in it, the combined code of both in B.incl can't exceed 32K.

- *This 32K limitation may be a thing of the past when the new PowerPCs arrive since they will use a different memory management technique.*

Visibility Restricted

An include file cannot see subroutines defined in another include file unless it's accessed with the INCLUDE statement in the first include file. When working on large projects, this can put almost as much of a crimp in program development as when we only had a single source file to play with.

The problem is that when the compiler tries to compile an include file, it must know where every subroutine used in the file exists in memory. Include files are by definition stand-alone in nature, and a subroutine referred to in the source code yet not found in the compilation process causes a major problem because that code is missing. As it is now, the compiler just complains that it can't find the referenced subroutine.

However, it's possible to overcome even this. For a method of circumventing this restriction, be sure to check out the Peak Performance section of this chapter.

Missing the Data

You can't use any DATA statements in an include file. One of the restrictions in the Macintosh design is that code resources can't use global data structures. FB will compile the include file without error, yet when it comes time to run the code, data is likely to be missing. This is most evident when using strings.

The solution is to store your data in custom resources. Or, if you must use DATA statements, always place them in the projects main source file.

Include File Tips

Include files are often misunderstood, let's take a moment to clarify some points. Each addresses a specific behavior that programmers often overlook.

- An include file is only as current as its last compilation.

When FB compiles the source in an include file, the compiled code contains a snapshot of the current state of all global files or include files called by the file. Any subsequent changes made to either the globals file or the nested include file will be invisible to the compiled version of the include file.

For example, if a global file is changed by adding, deleting, or re-arranging its list of records, variables or arrays, the include file that calls it with a GLOBALS statement won't know of the changes until it has been recompiled. Plus, if an

include file itself calls another include file and you make changes in the nested include, the original include won't see the changes until its been recompiled.

- Compile both the main and all include files using identical COMPILE statement setting.

One of the fastest ways to create quirky bugs in your programs is to mix and match COMPILE statements among the main and all its include files. If the main file uses:

```
COMPILE 0, _strResource_macsBugLabels
```

Then be sure to use the same statement in all includes called by the main program. Like global files, it ensures that all modules in the project play with the same sheet of global music.

One of the most common COMPILE mismatch errors is to use _caseInsensitive in one file and not in another. This creates the very annoying "function not found" error. However, it's easy to fix by matching the COMPILE statements of all concerned files.

It is especially important to watch out for mixing of the following compiler settings: _caseInsensitive, _arrayBase1 or _arrayBase0, _optimizeAsInt and _dontOptimize, and _chkRuntimeErr and _noRuntimeErrs. Any mismatch of these settings can create subtle problems that are hard to locate.

- Always use the _strResource setting.

As previously mentioned, code resources are not allowed to have any global data statements. To overcome this restriction, use DATA statements in the main program, or for strings, use the _strResource setting of the COMPILE statement. This ensures that the compiler always saves your code resource strings with the code itself.

Regular Exercise

Now that we understand globals and include files a bit better, let's convert *SimpleBase* into a program that uses them.

Adding a Global File

Adding a global file is pretty easy. The steps are:

1. Copy all of the global declarations created with DIM statements from the .main document to the Clipboard.
2. Choose **New** from the **File** menu to create a blank source file.

3. Paste the Clipboard contents into the new file.
4. Save the file using the `.glbl` suffix for easy identification. We call ours *SimpleBase.glbl*.
5. Return to the `.main` source file. Replace all of the global declarations with `GLOBALS "SimpleBase.glbl"` just before the `END GLOBALS` statement.
6. Save the changes made to the `.main` file.

Try out the new version of *SimpleBase*. Notice any difference? You shouldn't.

Adding Include Files

We are going to create two include files for use with *SimpleBase*. The first will be a universal include that contains all of our button, edit field, and cursor event handlers. We separate these routines out because they are often used, and as a separate file they will be much easier to add to other projects later. The second include file will contain the rest of our program code. Ready? Let's get started.

DialogEvents.Incl

To add the universal include that contains all of the common edit field, button, and cursor handling subroutines, follow these steps:

1. Choose **New** from FB's **File** menu.
2. Add the following lines to the new untitled document:

```
INCLUDE FILE _aplIncl
GLOBALS "SimpleBase.glbl"
END GLOBALS
```

3. Save the file as *DialogEvent.Incl*
4. Open the *SimpleBase* file.
5. Copy the following subroutines from the *SimpleBase* file to the *DialogEvent.Incl* file:

```
FN CursorHandler
FN EFClickEvent
FN TabShiftTabEvents
FN CheckBoxHandler%
FN RadioBtnHandler%
FN HiliteSelectedButton
FN ChangeReturnToBtn
```

6. Save the *DialogEvent.Incl* file.

7. Choose **Run** from the **Compile** menu. If everything is correct, the editor will translate the source into compiled code within the include file.

8. Return to the *SimpleBase* source file and remove all of the previously mentioned subroutines. Save your changes.

That's it. We now have an include file that we can use in any program to handle some common dialog events. To see how we call it, continue with the next section.

SimpleBase.Incl

The biggest change is with moving the majority of our subroutines from the main source file to an include. The following steps describe exactly how to do this:

1. Choose **New** from FB's **File** menu.

2. On the new untitled document add the following lines:

```
INCLUDE FILE _aplIncl
GLOBALS "SimpleBase.glbl"
END GLOBALS
INCLUDE "DialogEvent.Incl"
```

3. Save the file as *SimpleBase.Incl*

4. Open the *SimpleBase* file.

5. Save as under the name *SimpleBase.main*.

6. Copy everything *BUT* the following subroutines from the *SimpleBase.main* file to the *SimpleBase.Incl* file:

```
FN BuildMenus
FN Initialize
Everything below the Main Loop section marker
```

7. Add the `INCLUDE` statement to *SimpleBase.main* at the specified location:

```
END GLOBALS
INCLUDE "SimpleBase.Incl" <-- ADD THIS LINE
' --- FUNCTIONS ---------------------------
```

8. Save your changes to both files.

9. Compile the *SimpleBase.Incl* file.

- *If you haven't already compiled the DialogEvent.Incl you will get an error. Just compile DialogEvent.Incl and try again.*

Now run *SimpleBase.main*. Does it compile a bit faster? It really doesn't, but since the majority of code is already compiled in *SimpleBase.Incl*, the main file flies when it comes time to run.

Peak Performance

As mentioned earlier, one include file can't look into the contents of another. At least, not without some help from us. In this section we will examine one technique of providing that access.

The Project Include

The method we will examine requires a specialized include file that I'll call *Project.Incl*. In essence, the *Project.Incl* file contains a listing of all the subroutines that must be globally available to the entire project, no matter which file they reside in. It does this by indirection.

The problem with compiling an include file that is trying to access a subroutine in another include, is that the compiler can't find the subroutine in the include. What we need to do is trick it into finding the subroutine. How? By providing the subroutine in the *Project.Incl* file. Here's how it works.

In the *DialogEvent.Incl* file are seven routines to handle common dialog events. Let's create a *Project.Incl* file that contains pointers to those subroutines. When we add the *Project.Incl* to another file, the routines in it become available to that file. If we add it to all of a project's include files, they are available to them all.

Creating a Project Include File

The code to create a project file for *Simplebase* is shown in Program 108. It makes use of the `FN USING` statement to redirect program control to another address in memory.

It starts like a standard include file with `INCLUDE FILE`, and accesses the project's global file with `GLOBALS`. After that it differs.

The next section contains all of the definitions for our globally available functions using `FN USING`. We use the same names defined earlier to call the original subroutines. Each `FN USING` statement defines the exact parameters used by the original function. For example, the function `CursorHandler` expects two parameters. Our project function definition must accept the same two parameter types. The names don't have to match, just the type.

Each function definition also uses a global pointer variable. Program control will be directed to the address stored in the pointer when the project subroutine is called. Since it's globally defined, FN USING doesn't care where the real subroutine resides, it just directs control to it.

PROGRAM 108. The Project Include.

```
INCLUDE FILE _aplIncl
GLOBALS "SimpleBase.glbl"
END GLOBALS
' --- GLOBAL PROJECT FUNCTIONS ---------------------
DEF FN CursorHandler (cursEvtID%,dlgID%) USING gCursorPtr&
DEF FN EFClickEvent  (fieldID%) USING gEFClickPtr&
DEF FN TabShiftTabEvents (dlgEvnt%,startFld%,lastFld%) USING gTabEventsPtr&
DEF FN CheckBoxHandler% (btnID%) USING gCheckBoxPtr&
DEF FN RadioBtnHandler% (lowBtnID%,highBtnID%,setBtnID%) USING gRadioBtnPtr&
DEF FN HiliteSelectedButton (btnID%) USING gHiliteBtnPtr&
DEF FN ChangeReturnToBtn (@evntIDPtr&, btnID%) USING gReturnToBtnPtr&
```

Revising DialogEvent.Incl

There are a couple of things that we need to do to the *DialogEvent.Incl*. We start by changing all of the function names to the project names. For simplicity, I just precede each function name with a lowercase "p" for project. Then, at the bottom of the file, add the lines of code shown in Program 109.

Essentially, these lines use the @FN statement to get the address of the specified subroutine and place them into the global values we used with the *Project.Incl* file. These lines are executed each time the program runs and they provide the critical link to access the project subroutines.

PROGRAM 109. Revised DialogEvent.Incl.

```
' --- SET PROJECT ADDRESSES ------------------------
gCursorPtr& = @FN pCursorHandler
gEFClickPtr& = @FN pEFClickEvent
gTabEventsPtr& = @FN pTabShiftTabEvents
gCheckBoxPtr& = @FN pCheckBoxHandler
gRadioBtnPtr& = @FN pRadioBtnHandler
gHiliteBtnPtr& = @FN pHiliteSelectedButton
gReturnToBtnPtr& = @FN pChangeReturnToBtn
```

Revising SimpleBase.glbl

Revising the *Simplebase.glbl* file just requires that we add the necessary global pointers defined in the *Project.Incl* file to it. This ensures that every file that uses the global file will see the pointers, and allow them to access the working subroutines stored at that memory address.

PROGRAM 110. Revised SimpleBase.glbl.

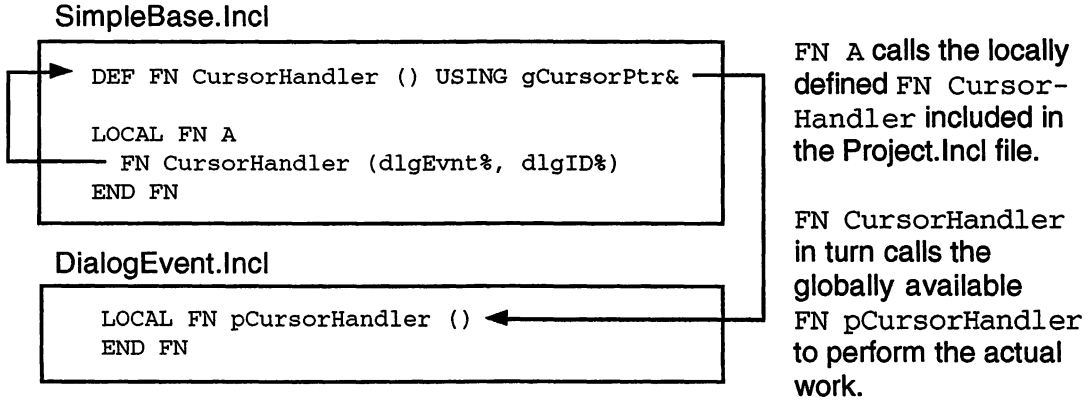
```
' --- DEFINE PROJECT ADDRESS POINTERS ---------------
DIM gCursorPtr&
DIM gEFClickPtr&
DIM gTabEventsPtr&
DIM gCheckBoxPtr&
DIM gRadioBtnPtr&
DIM gHiliteBtnPtr&
DIM gReturnToBtnPtr&
```

Understanding Project Includes

So what happens when we make a call to a project available function? The entire sequence is graphically shown in Figure 63.

Each include file has added the *Project.Incl* file to it, and each makes a call to its locally defined `FN USING` statement. The `FN USING` statement accepts the parameters required by the subroutine that does the work, and then passes control to the address specified by its global pointer value. At that address (which can be anywhere in program memory) is the working subroutine. The working subroutine does its thing with the parameters, then returns control to `FN USING`, which passes control back to the calling routine. To the calling routine it looks as if the original project subroutine performed the task.

FIGURE 63. The calling sequence.



Multiple project include files are possible within a single project. It's even better to have several small project includes instead of one single humongous one. Why? Because each project include is added to *EVERY* file that uses it, the code can get duplicated many, many times. By keeping the project include small, your other include files can have room to grow without feeling cramped in their 32K limit.

Cooldown

In this chapter we discussed ways of extending a program from one file to many. We started with describing global files and how to implement them in all of your programs. We followed globals by describing include files, and their strengths and weaknesses. We showed how it's possible to use them to increase the pace of program development and allow you to re-use code in different projects. Finally, in Peak Performance we showed how to access subroutines in different include files using a *Project.Incl* file.

All-in-all, a busy chapter, but one that will serve you well as your programming projects grow in size and complexity.

CHAPTER 13

Resources

Warm-up

This chapter introduces you to resources. In it you will learn:

- What resources are,
- How to use ResEdit to create resources,
- What pointers and handles are,
- How to implement resources in your programs, and
- How to manipulate external resource files.

What are Resources?

A **resource** is a structured form of data stored in the resource fork of a file.

Resources are grouped into resource types. **Resource types** are 4-character alphanumeric identifiers that uniquely identify a resource's data structure. Some common resource types include `CODE`, `MENU`, `CURS`, `DLOG`, `WIND`, etc. These resource types are known as **standard resources** since their format has been defined by Apple. You are not restricted to using standard resources, instead, you can also design custom resources for your programs.

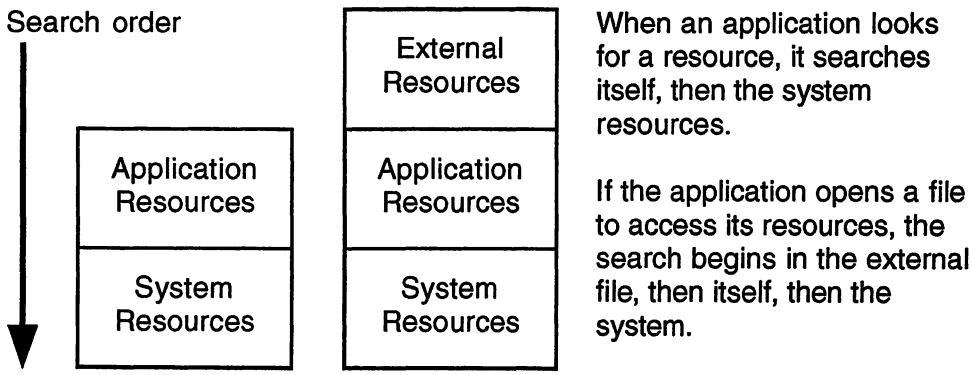
Remember that a file can consist of a data fork, a resource fork, or both. Normally, applications consist entirely of resource forks while program data files are stored in file data forks. There are, however, exceptions to every rule. A look at the *FutureBASIC Extras* file shows that it's made up entirely of resources. Regular source code files saved as text are exclusively data.

Source files that have been compiled as include files contain both data and resource forks.

A program can have any number of resource files open at one time. The **system resource file** is always the first one opened by an application and consists of all resources stored in the System file. The next resource file opened by a program is its own **application resource fork**. Additional resource files can be opened under both program and user control depending on the nature of the program.

Resource files are always searched in the reverse order of their opening. An application that has opened an external resource file will look for a resource in: the external resource file, the application, and finally the system resources. There are some Toolbox calls that enable you to manipulate this search order. We'll see how to use them later in the chapter.

FIGURE 64. Resource file search order.



You access resources using Toolbox calls. A resource can be referenced using a name, an ID number, or its type and index number. Many common FB commands allow you to open resources easily. Examples include the `MENU`, `SOUND`, and `STR#` statements. Other resource types can be accessed using standard Toolbox routines like `FN GetResource`.

Resources have attributes associated with them. **Resource attributes** determine how the resource will be dealt with by the Resource Manager. These are described in Table 10.

TABLE 10. Resource attributes.

Attribute	Explanation
_resPurgeable	Marks the resource as purgeable from memory should the Memory manager need to make room.
_resProtected	Prevents the resource from being changed by other resource modifying calls.
_resChanged	Marks the resource as changed.
_resPreload	Marks this resource for loading immediately after the resource file it resides in is opened.
_resSysHeap	Identifies which heap (application or system) the resource resides in.
_resLocked	Locks a resource in memory so that it can't be purged.

ResEdit

The most often used resource editing program is *ResEdit*. Originally designed for internal program development at Apple, it has since been distributed worldwide for the benefit of programmers everywhere. While it does have a quirky interface, and lacks some useful features, it provides a good overall capability for creating and designing resources both standard and custom resources.[1]

Understanding *ResEdit* is an entire journey in and of itself. There is not enough room in this book to detail all the ins and outs of *ResEdit*. If you have additional questions, check out the bibliography in the back of the book.

I can only give you a few guidelines and point out the most hazardous areas, but the major warning I give is this:

Danger, Will Robinson!
Always work on a copy of the file you are modifying.

I can't say this enough. Because *ResEdit* works in strange and mysterious ways to manipulate resources, you court trouble working on any original file.

1. An even more full featured resource editor called Resorcerer® can be obtained from Mathemæsthetics, Inc. at 617.738.8803.

A power fluctuation, a cat running across your keyboard, an errant key press, a spilled drink, all of these and more can cause you to lose an entire file you've just spent hours editing. One error in any resource and it's bomb city the next time you attempt to use it. So be careful.

If you forget to do this, don't say I didn't warn you.

A Quick Tour

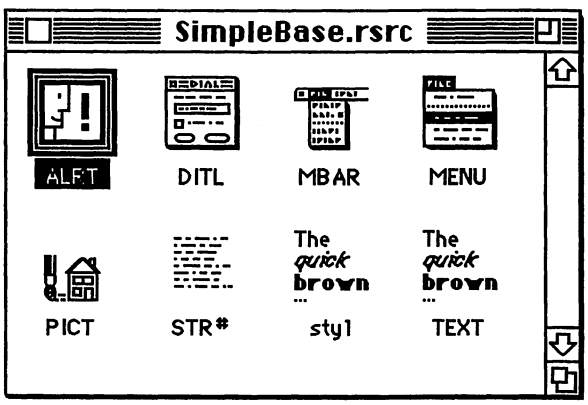
For a quick tour of *ResEdit* you'll need to have it loaded on your disk. It's available on disk #3 of the *FutureBASIC* package. If you haven't already done so, take a minute and load it onto your hard disk. Come back when you're ready after booting up *ResEdit*.

To open a resource file in *ResEdit*, select **Open** from the **File** menu. *ResEdit* will display all files and folders on your disk, including invisible files in the open file dialog. Select the file you wish to view and click the **Open** button.

While *ResEdit* will attempt to open any file, it can't directly edit files that only contain a data fork[2]. *ResEdit* will ask if you want to add a resource fork to any file that doesn't have one. In most cases you should say "No" unless you really mean to add resources to the file.

Once a file is open, you'll see a resource picker window like that shown in Figure 65 displaying all of the resource types in the file. A **picker window** is simply a window that enables you to view and select resources within the file in icon format. Use the **View** menu to display any of the picker windows in either icon or type order.

FIGURE 65. Application resource picker window.

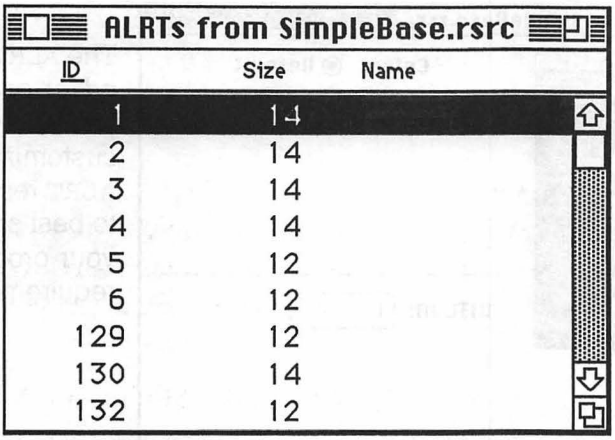


The application picker window displays all of the resource types currently present in the resource fork of the open file or application.

Double click any icon to open that resource's picker window.

2. Resorcerer® allows you to edit both the resource and data forks of a file.

FIGURE 66. ALRT resource picker window.



The resource picker window shows all of the resources of the indicated type.

Shown in this picker window is the ID number, size, and name of ALRT resources.

To view a particular resource type, double-click on the icon for that resource type. For example, a double-click on the ALRT icon opens the ALRT resource picker window shown in Figure 66. As you can see, there are several individual ALRT resources in this file, each identified by a unique resource ID and shown with its size and name. A **resource ID** is an integer value used to identify a certain resource of the specified type. Again, you can change this picker view to display the resources in icon, ID, name, or order in the file. A double-click on any particular resource ID will open the ALRT custom editor.

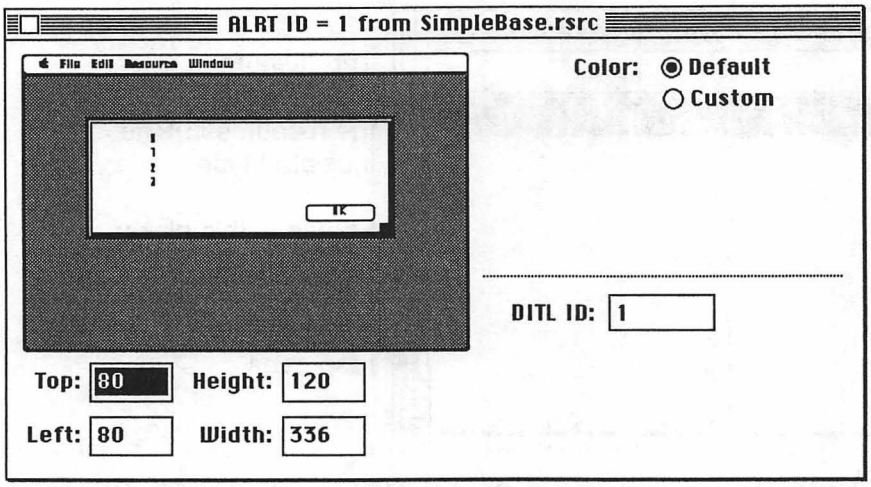
Custom Editors

ResEdit wouldn't be of much use if you couldn't easily edit certain resource types, so it contains over 60 custom editors. These allow you to create and manipulate such diverse resources as MENU, WIND, DLOG, DITL, PICT, ALRT, cicn, ICN#, CURS, STR, STR#, TEXT, styl, and a host of others. Custom editors have been created both by Apple and by resourceful people in the programming community who just couldn't wait for Apple to release more.

For example, by double-clicking on ALRT ID 1 (a standard ALRT resource supplied by FB), it opens the custom ALRT editor window shown in Figure 67. Note that it allows customizing of the ALRT resource including its size, its color, and which DITL resource it uses for its items. In addition, you can click on the alert shown in the small mini-screen within the window and move or resize it with the mouse.

A double-click on the small ALRT in the mini-screen will open the DITL (dialog item list) custom editor which allows you to move, add, rename, or delete the actual items in the ALRT resource. The DITL editor can be seen in Figure 68.

FIGURE 67. Custom ALRT editor window.

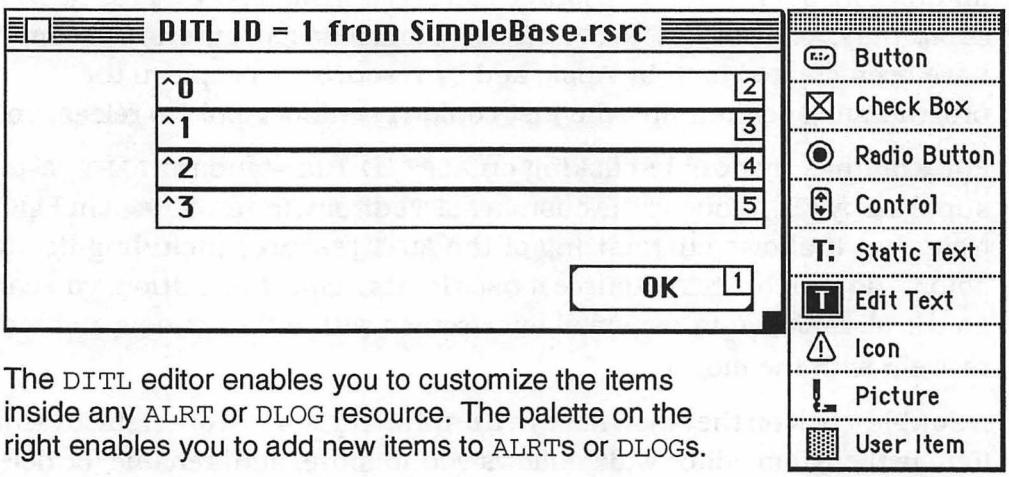


The ALRT editor enables you to customize any ALRT resource to best suit your program's requirements.

Note the floating palette containing the items that can be placed in an ALRT or DLOG resource.

ResEdit has many custom editors. We'll cover most of the common ones in later chapters as we deal with specific resources. However, there is a general order you can follow that enables you to create resource files and add any type of resource your program needs. Let's see how to do that.

FIGURE 68. Custom DITL editor for ALRT #1.



The DITL editor enables you to customize the items inside any ALRT or DLOG resource. The palette on the right enables you to add new items to ALRTs or DLOGs.

Creating Resources

Once you have a resource file open, it's time to create a program resource file. Create a new resource file by choosing **New** from the **File** menu. At the save dialog, enter the name for the file, in our case "*SimpleBase.rsrc*", and click on the **Save** button. *ResEdit* creates the resource file in the chosen folder, then displays an empty resource picker window.

Once we have a resource file open, it's simple to add any type of resource a program requires. Start by choosing **Create New Resource** from the **Resources** menu. The resource type window that appears shows a scrolling list of all the templates for resource types *ResEdit* can create. Click on a resource type, or enter its 4-character specifier, then click **OK**. *ResEdit* will open its picker window, create a resource of the requested type, and open its editor for you.

Deleting Resources

Deleting resources once they have been created is easy. Simply select the individual resource to delete and choose **Clear** from the **Edit** menu. You can delete individual resources from within the resource picker window, or an entire resource type from within the application picker window.

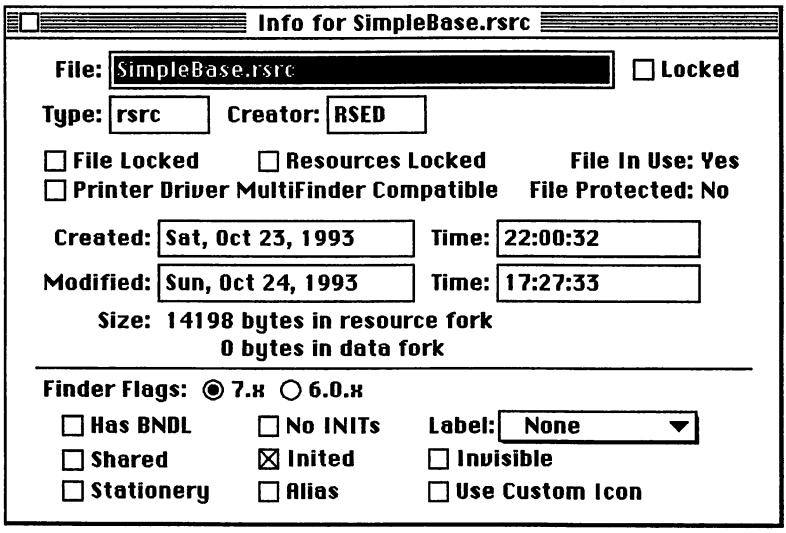
Remember that some resources have partners. For example, both `ALRT` and `DLOG` resources have `DITL` resources associated with them, `TEXT` resources normally have `styl` resources attached to them. When deleting an `ALRT` or `DLOG` resource be sure to delete the orphan `DITL` resource too. It's not hard to locate orphaned `DITL` resources, they'll have the same ID number as the `ALRT` or `DLOG` resource just deleted. `TEXT` resources will also have a `styl` resource with a matching ID number.

Getting File Information

Another feature *ResEdit* has is the capability of viewing and changing both file and folder information. This information is normally only accessible to your programs using either `GET FILE INFO` or the Toolbox function `GETFINFO`. You can view this information by selecting **Get File/Folder Info...** from the **File** menu and opening a file or folder, or by selecting **Get Info for This File** (where This File is replaced by the currently active file name) from the **File** menu.

For example, in Figure 68 we can see the file information from a sample file. It contains the file name, status, its type and creator, some special system settings, the file creation and modification dates, the size of both file forks,

FIGURE 69. File information dialog.



and the special Finder flags. Normally, the System takes care of handling all of these settings so you don't have to. The only ones we'll play with are the Type and Creator fields. We'll see how to do that in a later chapter.

Pointers and Handles

One of the common things you hear about when dealing with resources is pointers and handles. While the actions of pointers and handles may at first seem overwhelming to the weekend programmer, it isn't.

Pointers

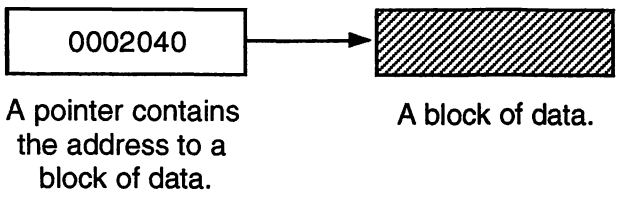
A **pointer** is a nonrelocatable block of memory that contains a memory address. The address contains the pointer data. Once created, a pointer never moves in memory, making it possible to locate the data simply by reading the pointer address.

Creating and Disposing of Pointers

You can create a pointer using the Toolbox command `FN NewPtr`. It accepts size value and returns a pointer to a non-relocatable block of memory of the size requested. For example, to create a pointer that will hold a RGB color record (6 bytes), do this:

```
rgbPtr& = FN NEWPTR (_rgbColor)
```

FIGURE 70. How a pointer works.



When finished with a pointer, be sure to dispose of it to free memory for other activities. To dispose of our color pointer do this:

```
osErr = FN DISPOSEPTR( rgbPtr&)
```

Accessing Pointer Data

You can use the standard PEEK LONG or PEEK WORD to read data from a pointer's data block. For example, read an integer value like this:

```
data% = PEEK WORD (dataPtr&)
```

or long integer data like this:

```
data& = PEEK :LONG (dataPtr&)
```

Or use the record reading method with a single dot and an offset like this:

```
data% = dataPtr&.none%
```

and this:

```
data& = dataPtr&.none&
```

Note that the constant offset _none equals zero, enabling us to read the first two or four bytes of data from the data block. By changing this field offset using a different constant (even ones unrelated to the actual data stored in the block) we can read further into the data block like this:

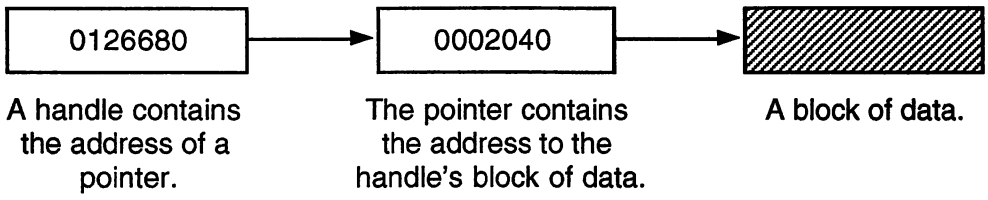
```
data& = dataPtr&.bottom&
```

Which returns a long integer six bytes into the pointer's data block.

Handles

A **handle** is a pointer to a pointer. In other words, a handle is a pointer (a memory address) that points to a non-moving master pointer (a second memory address) that in turn points to the data. Since a handle's block of data is relocatable, the Memory Manager can move location at certain well defined times. Whenever it moves the data, it updates the master pointer with the new address, ensuring that it always points to the block of data.

FIGURE 71. How a handle works.



Creating and Disposing of Handles

You can create a handle using the Toolbox command FN NewHandle. It accepts a size value and returns a handle to a relocatable block of memory of the size requested. For example, to create a handle to a RGB color record (6 bytes), do this:

```
rgbH& = FN NEWHANDLE (_rgbColor)
```

When finished with a handle, be sure to dispose of it to free up memory. To dispose of our color handle we can do this:

```
osErr% = FN DISPOSEHANDLE (rgbH&)
```

Accessing Handle Information

You can retrieve information from a handle using a couple of methods. First, you can manually use PEEK LONG to navigate the handle and pointer addresses, like this where:

```
dataPtr& = PEEK LONG (myHandle&)
```

Returns the data address stored in the pointer. This method of accessing handle data via the pointer is known as **de-referencing**. For example, to read an integer from the handle's data block do this:

```
data% = PEEK WORD (dataPtr&)
```

and:

```
data& = PEEK LONG (dataPtr&)
```

returns a long integer. We can adjust the offset from the data block's starting address to read further into the data like this:

```
data& = PEEK LONG (dataPtr& + offset&)
```

An alternate method for reading handle data is to use the double-dot record reading format and let the runtime do the work. This enables us to read the data directly without confusing ourselves with PEEK LONGs and PEEK WORDs. For example, this returns an integer:

```
data% = dataHandle&..none%
```

And this returns a long integer from the data block:

```
data& = dataHandle&..none&
```

Using Program Resources

The method of choice for accessing our program's resource file is to use the RESOURCES statement. RESOURCES allows us to use our program's resource file from within the editor during program development, and neatly bundles the file with our final application when we build it.

The RESOURCES statement will do the following for us:

In the editor while developing a program it:

- Opens the resource file for quick access by the program, and
- Closes it when we quit testing our program

When building an application it:

- Copies all program resources to the final application,
- Assigns the creator and file type specified to the final application, and
- Ensures that a SIZE resource is added to the application

It's the RESOURCES statement that makes it easy to use *ResEdit* to create program resources like alerts, dialogs, strings, icons, pictures, etc. and use them from within the editor during program development and testing. All it needs is the name of the resource file to use:

```
RESOURCES "SimpleBase.rsrc"
```

Additionally, the resource file must be in the same folder as the program's source code. Since most projects are naturally grouped into a folder to use the Project Manager tool, this shouldn't present a problem.

Using External Resources

Using resource files external to the application is a bit different. There is no single statement that handles all the complexities of dealing with external resource files. An external resource file is one that will be opened by a program and its resources read, updated, deleted, or added as required. *ResEdit* is the epitome of this – it's sole purpose is to manipulate the resources of other files.

When it comes to dealing with resource files the only way to do so is using the Macintosh Toolbox. While FB makes it easy to use sound, pictures, strings, and icons with built-in commands, accessing other resources requires Toolbox commands from the Resource Manager.

Opening & Closing Resource Files

Opening an external resource file is made simple by using USR OPENRFPERM. This function accepts the same parameters returned by FILES$, a filename and a WD reference number, as well as a permission setting. In turn, it returns a resource reference number that is used to manage the resource file while it's open. For example:

```
rsrcRef% = USR OPENRFPERM (filename$, wdRefNum%, perm%)
```

Will open the specified file's resource fork. The types of file access permission include: read only (_fsCurPerm), write only (_fsWrPerm), exclusive read & write (_fsRdWrPerm), shared read & write (_fsRdWrShPerm), and whatever permission is allowed (_fsCurPerm). Normally, I use _fsCurPerm unless I have a specific objective in mind, like listing all DLOG resources in the file. Then I would use _fsRdPerm instead.

When finished with a resource file, close it using the resource reference number obtained from USR OPENRFPERM like this:

```
CALL CLOSERESFILE (rsrcRef%)
```

Managing Several Resource Files

We mentioned earlier that an application can open multiple resource files. And since we need to search them to open specific resources, it's obvious that we need some commands to allow us to manage those open files.

Why do we need to do this? Well, since each resource file contains its own list of resources, its possible that many of them may conflict in resource ID. For example, imagine that we want to open the PICT resource in FB containing its about picture which happens to have a resource ID of 258. But what if the System has an identically numbered PICT resource? Both resource forks are open. Which PICT resource should the Resource manager open?

The answer is, whichever one it finds first. If the application's resource fork is searched first, the correct about picture will be shown. However, if the System is searched first, the wrong one will appear. As you can see, it can be very important which resource file is accessed. The following commands will help you do that.

CurResFile

A method of managing several resource files is to use the rsrcRef% associated with each resource file. For example, to get the resRefNum% of the current resource file use FN CurResFile like this:

```
currentResRefNum% = FN CURRESFILE
```

UseResFile

Since every time we open a resource file we get a `rsrcRef%` in return, we can use that to manage the order of our resource search. For example, if we need to search the System resource fork before the applications, just do this:

```
systemResRefNum% = {_sysMap}
CALL USERESFILE (systemResRefNum%)
```

HomeResFile

To locate the resource file where a particular resource comes form, use `FN HomeResFile`. When given a handle belonging to a resource, it returns the resource's file `rsrcRef%`. Assuming that we have a valid resource handle, do this:

```
myResRefNum% = FN HOMERESFILE (resourceH&)
```

Creating a Resource File

Creating a file with a resource fork is something you might need to do. This task falls to the Toolbox call `CreateResFile` and is implemented like this:

```
CALL CREATERESFILE (fileName$)
```

It creates a blank resource file containing no data in its resource fork. Once our file has a resource fork, the commands in the next section allows you to add, delete, or change the resources inside.

Adding & Deleting Resources

There is only one command for creating a resource and that's `FN AddResource`. `FN AddResource` requires four pieces of information from us in order for it to successfully add a new resource to a file. It requires a handle to the data to store the resource in, a resource type, a resource ID, and optionally, a name. For example, if we had a handle it would be possible to save it as a resource like this:

```
CALL ADDRESOURCE (hndl&, resType&, resID%, resName$)
```

Where `resType&` is any 4-character alphanumeric converted to a long integer. It's possible to do this using:

```
resType& = CVI("PICT")
```

or use the shorthand method of:

```
resType& = _"PICT"
```

This can be combined in the `AddResource` call like this:

```
CALL ADDRESOURCE (hndl&, _"PICT", resID%, resName$)
```

RmveResource

When it's time to remove a resource, use the procedure RmveResource. All that it requires is a valid resource handle as an argument. For example, to dispose of a PICT resource do this:

```
aboutH& = FN GETPICTURE (258)
CALL RMVERESOURCE (aboutH&)
```

ReleaseResource

The ReleaseResource command enables you to dispose of a resource handle, freeing up the memory it occupied. Don't use this unless you are absolutely sure you're done with the resource. To release a resource just pass it a resource handle like this:

```
CALL RELEASERESOURCE (resH&)
```

DetachResource

Finally, we come to DetachResource. You will find this command very useful with resources. Since a resource is linked to its resource file, you can't use the resource if the file isn't open. However, if you detach the resource's handle from the associated file, you can safely close the file and continue to use the resource. To detach a resource from its resource file do this:

```
CALL DETACHRESOURCE (resH&)
```

Getting Resources

Many of the standard resource types have their own calls to access them from a resource file. We'll see how many of them work in later chapters. However, in this chapter we'll deal with one that *SimpleBase's* employee record must deal with, pictures. The Toolbox call we favor is FN GetPicture. But there are several other methods of accessing a specific resource: by resource type and ID, by type and index position, and type and name.

GetPicture

FN GetPicture returns a handle to the specified PICT resource. A PICT resource contains a recorded series of QuickDraw commands to reproduce any image you might want to display. For example, to open FB's about picture and display it on the screen we can just do this:

```
pictH& = FN GETPICTURE (258)
PICTURE (10,10), pictH&
```

GetResource

For more general resources, we can use FN GetResource. FN GetResource requires a resource type and a resource ID in order to locate the correct resource. To open the same about picture with GetResource, do this:

```
pictH& = FN GETRESOURCE (_"PICT", 258)
PICTURE (10,10), pictH&
```

GetIndResource

However, if we absolutely knew that the PICT resource we wanted was the fifth of seven PICT resources we could also do this:

```
pictH& = FN GETINDRESOURCE (_"PICT", 5)
PICTURE (10,10), pictH&
```

GetNamedResource

And finally, if the PICT resource has a name associated with it we could have done this:

```
pictH& = FN GETNAMEDRESOURCE (_"PICT", "FB About")
PICTURE (10,10), pictH&
```

Regular Exercise

We just covered a lot of information on resource files and resources. Now let's add some resource file support to *SimpleBase*.

Before we begin, we need one global variable. This variable will contain the handle of the current employee picture being displayed. Add the following line to the *SimpleBase.glbl* file:

```
DIM gPictH&
```

The employee pictures themselves will be saved in the resource fork of the employee file as common PICT resources. Each PICT resource will have a resource ID that matches the employee record number. I.e., employee record #1 will use a PICT resource with an ID of 1 also.

Creating an Employee File

Out first order of business is to add a resource fork to our newly created employee file. To do that, we modify the DBNewEmployeeFile subroutine as shown in Program 111 to first open a blank file with the standard OPEN statement, close it, then add a resource fork to the same file using CreateResFile. That's all there is to it. The rest of the subroutine remains the same.

PROGRAM 111. Revised DBNewEmployeeFile subroutine.

```
LOCAL FN DBNewEmployeeFile
   FN DBBlankRecord
   DEF OPEN "SbDbFbSb"
   OPEN "O", #1, gFileName$, , gWDRefNum%
   CLOSE #1
   CALL CREATERESFILE (gFileName$)
   gOpenRecord% = 0
   gEmployee.dbName$ = "Created by SimpleBase, from the book:"
   gEmployee.dbAddr$ = "Learning FutureBASIC: Macintosh BASIC Power"
   gEmployee.dbCity$ = "By Sentient Fruit™"
   FN DBWriteRecord
   INC (gOpenRecord%)
END FN
```

Reading Employee Pictures

The routine to read our employee pictures is called `ReadEmployeeGraphic` and can be seen in Program 112. It begins by using `USR OPENRFPERM` to get a resource reference ID to the resource fork of the employee record. If the ID is valid, we next check to see if the currently open employee record has a picture associated with it. If the employee record does have a matching picture, the subroutine disposes of the previous picture handle with `DEF DISPOSEH`.

Next it uses `FN GetPicture` to read the employee's picture into memory and check to make sure we got it. It uses the `EDIT$` function to display the picture in the picture field of the Data Entry window. Then, it detaches the picture resource from the employee file with `Detachresource` and closes it using `CloseResFile`.

Finally, we add calls to the `ReadEmployeeGraphic` from both the `RecordFieldToEF` subroutine and the `_wndRefresh` section of the `DialogEntryWindow` function.

Adding Employee Pictures

When it comes time to add an employee picture to an employee file we use the `SaveEmployeeGraphic` shown in Program 113.

It begins by opening the employee file with `USR OPENRFPERM`. If it has a valid resource reference number, it checks to see if a `PICT` resource already exists using the `gOpenRecord%` ID number. If one does, it deletes it from the employee file. This ensures that the latest image is always stored to disk.

PROGRAM 112. ReadEmployeeGraphic subroutine.

```
LOCAL FN ReadEmployeeGraphic
  resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, _fsCurPerm)
  LONG IF resRef% <> _nil
    LONG IF gEmployee.dbPictID% > _nil
      DEF DISPOSEH (gPictH&)
      gPictH& = FN GETPICTURE (gEmployee.dbPictID%)
      LONG IF (gPictH& <> _nil) AND (FN RESERROR = _noErr)
        EDIT$ (_dbPhotoFLD) = &gPictH&
        CALL DETACHRESOURCE (gPictH&)
      END IF
    END IF
    CALL CLOSERESFILE (resRef%)
  END IF
END FN
```

Next, it adds the picture handle stored in the global variable `gPictH&` using `AddResource`, then sets its purgeable bit using `SetResAttrs`. It's marked as changed with `ChangedResource`, and `WriteResource` immediately saves it to disk. We detach the new resource from its home file using `DetachResource` and close the employee file.

Finally, we add a call to the `SaveEmployeeGraphic` from the `EFtoRecord-Field` subroutine.

PROGRAM 113. AddPicture2File subroutine.

```
LOCAL FN SaveEmployeeGraphic
  resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, 0)
  LONG IF resRef% <> 0
    LONG IF gEmployee.dbPictID% > _nil
      tmpH& = FN GETPICTURE (gOpenRecord%)
      IF tmpH& THEN CALL RMVERESOURCE (tmpH&)
      CALL ADDRESOURCE (gPictH&, _"PICT", gOpenRecord%, "")
      CALL SETRESATTRS (gPictH&, _resPurgeable%)
      CALL CHANGEDRESOURCE (gPictH&)
      CALL WRITERESOURCE (gPictH&)
      CALL DETACHRESOURCE (gPictH&)
    END IF
    CALL CLOSERESFILE (resRef%)
  END IF
END FN
```

Peak Performance

Resources are a Macintosh creation and their diversity and usefulness can't be over emphasized. What follows is a method of translating those old DATA statements into modern custom resources.

DATA Arrays

Many people still insist that the only safe means of storing static program data is in DATA statements. This method does have some benefits, like the ability to modify them directly in the program's source code and it's a bit easier to understand. But, the drawback of this method is that the arrays themselves can waste vast amounts of static memory, even if they're never used!

What's the solution? Move the information into custom resources. The benefits include: your information will only occupy memory when you need it, it's flexible, and it can be read into and saved out to disk much faster.

For demonstration, let's imagine that we have an integer array of 10,000 elements. We need these pre-calculated values to calculate the trajectory of a falling leaf. By having them in memory we can speed up the time it takes to calculate a solution. In the old days, we might have been done something like this:

```
DATA 10, 5, 6, 3, 23, 89, 97, 3001, 89, 1002
DATA 23, 45, 67, 34, 56, 32, 53, 102, 76, 432
```

and so on for 10,000 values. When it came time to execute the program, the user waited while all of those DATA statements were read into an array like this:

```
FOR count% = 1 TO 10000
  READ leaf%(count%)
NEXT count%
```

Which essentially took the information stored in the DATA statements and made it useful to the programmer in an array.

Custom Resources

Now let's take that same array and save it as a resource. To start, read the information into the original array just like before, only this time, we're going to grab it and store the entire array into a resource using the ArrayTo-Resource function shown in Program 114.

PROGRAM 114. ArrayToResource subroutine.

```
LOCAL FN ArrayToResource (@arrayPtr&, arraySize&, resType&, resID%)
  resH& = FN NEWHANDLE (arraySize&)
  LONG IF (resH& <> 0) AND (SYSERROR = _noErr)
    osErr% = FN HLOCK (resH&)
    BLOCKMOVE arrayPtr&, [resH&], arraySize&
    osErr% = FN HUNLOCK (resH&)
    CALL ADDRESOURCE (resH&, resType&, resID%, "")
    CALL SETRESATTRS (resH&, _resPurgeable%)
    CALL CHANGEDRESOURCE (resH&)
    CALL WRITERESOURCE (resH&)
    CALL RELEASERESOURCE (resH&)
  END IF
END FN
```

The `ArrayToResource` function accepts four parameters, a pointer to the array to store, it's calculated size in bytes, a resource type, and a resource ID number. With these it creates a handle using `FN NewHandle`, locks it from moving, and then uses `BLOCKMOVE` to transfer the array to the handle. Then, it unlocks the handle, and uses `AddResource` to store it as a resource. Finally it makes sure the resource is made purgeable, marks it as changed, writes it to disk, and releases it from memory. For example, to save our leaf array we might have called it like this:

```
FN ArrayToResource (leaf%(0), 20000, _"leaf", 128)
```

Then, when it comes time to read it into memory again, just use the `ResourceToArray` function. It does exactly the opposite. It gets the described resource with `GetResource`, locks it, and uses `BLOCKMOVE` to copy the handle contents to the array. It then unlocks and releases the handle from memory.

PROGRAM 115. ResourceToArray subroutine.

```
LOCAL FN ResourceToArray (arrayPtr&, resType&, resID%)
  resH& = FN GETRESOURCE (resType&, resID%)
  LONG IF (resH& <> 0) AND (FN RESERROR = _noErr)
    osErr% = FN HLOCK (resH&)
    BLOCKMOVE [resH&], arrayPtr&, FN GETHANDLESIZE (resH&)
    osErr% = FN HUNLOCK (resH&)
    CALL RELEASERESOURCE (resH&)
  END IF
END FN
```

As you may have guessed, using this technique makes loading in a large array nearly effortless. Combine this technique with the `XREF@` statement to link a handle with an array structure, and you have the makings of a truly dynamic array.

Cooldown

Well, while that finishes our initial discussion on resources, our tour is far from over. In this chapter we learned what resources are, saw what they're made of, learned how to create them, read them, and save them. We learned how *ResEdit* helps us to create, design, and manage our program resources.

We also went over pointers and handles and why they're important. Then, we learned how easy it was to add a resource fork to any file, and how to safely add and read `PICT` resources from our employee file.

With all this new resource knowledge, it's time to look at some other resource types, and see how to create and use them in *SimpleBase*.

CHAPTER 14

Alerts

Warm-up

This chapter introduces alerts. In it you will learn:

- What alerts are,
- How to use the four standard alerts,
- How to create custom alerts, and
- How to implement alerts in programs.

What are Alerts?

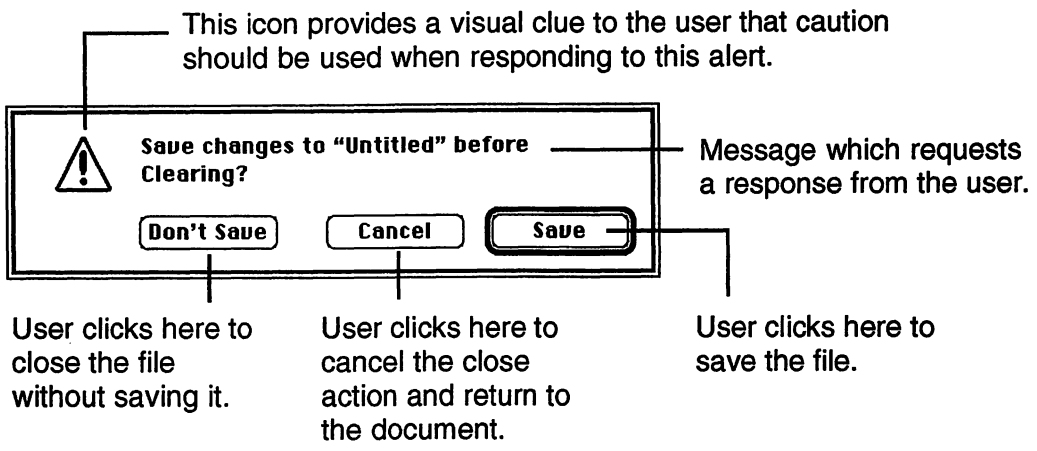
Alerts are special windows that contain informative text, controls, icons, and pictures. They normally report an error or provide warning to the user of a possible problem. Alerts can also play a sound or contain a message that requires a user's acknowledgment before a program takes action.

A common alert in FB is the alert that appears when the user tries to close a file that hasn't been saved. The save alert can be seen in Figure 75.

As you can see, this alert contains all of the features previously mentioned. It has a text message, displays an icon, and contains several buttons that the user can choose among. The user's response determines the fate of the file in question.

Alerts normally provide a default button (identified by a 3-pixel shadow) which indicates the default desired action for the alert. The default button should never encourage users to execute actions which may cause a loss of

FIGURE 72. FutureBASIC's standard save alert.



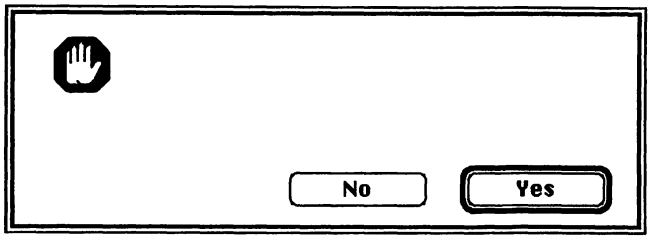
data. Many users seldom read the text of an alert, so a default button that would perform an irreversible action is not a good idea.

For example, the save alert above has a default button which saves the file's data. If instead, the default button was the **Don't Save** button, it might cause immense anxiety to the user when they throw away a file containing a full days work. Always err on the side of data safety and your users will love you.

TABLE 11. Alert function calls

Icon displayed:	Calling function:
none	`FN ALERT (alrtID, 0)` The simplest alert call, it shows an alert box without any specific icon.
	`FN NOTEALERT (alrtID, 0)` The note alert is used to provide information to the user or ask a simple question with multiple responses possible (Yes or No).
	`FN CAUTIONALERT (alrtID, 0)` Use the caution alert when the possible action may have undesirable results (such as losing data) if it's allowed to continue.
	`FN STOPALERT (alrtID, 0)` Use a stop alert when a problem or situation is so bad that the requested action cannot be completed.

FIGURE 73. Standard stop alert.



Default Alerts

There are four alert icon variations possible simply by using different Toolbox calls. These four alert functions are described in Table 11.

The default ALRT resources shown in Table 12 are always bundled with a compiled application. This enables you to use them during program development and still take advantage of them in the final application without having to re-create them. Each is designed to provide for a common alert situation requiring a response from the user, be it a simple acknowledgment or multiple choice option.

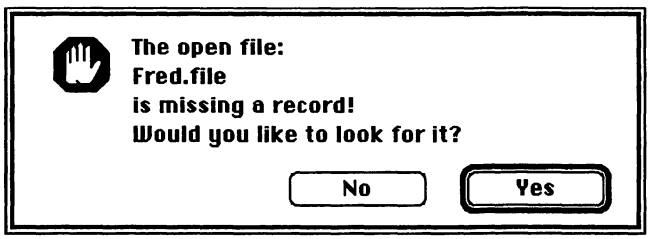
Each alert has a different default button to tailor the alert to a specific situation. A simple notice will probably need just an **OK** button. A dangerous alert situation may require both an **OK** and **Cancel** button to give the user a choice of actions.

For example, to display ALRT resource #3 as a STOPALERT you would do this:

```
item = FN STOPALERT (3, 0)
```

which is shown in Figure 75 when executed.

FIGURE 74. Standard alert with custom text.



Adding custom text to the alert allows you to customize the alert for multiple situations without bulking up your application with even more alert resources. You add text to an alert (or dialog) using the PARAMTEXT call. CALL

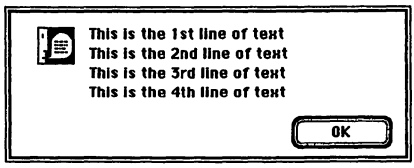
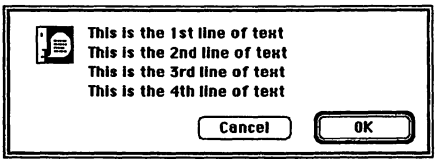
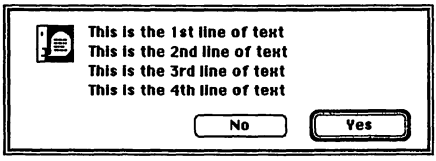
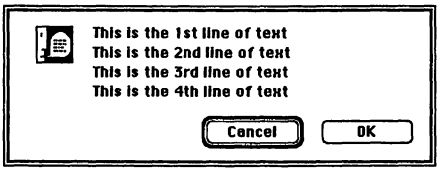
PARAMTEXT accepts up to four string variables (maximum of 63 characters each) and places them into global memory.

When displayed, the alert checks its static and editable text fields to see if they contain any ^n symbols (where *n* ranges from 0 to 3 exclusively). If one is found, the alert then replaces each ^n with the appropriate string. String one always replaces the ^0 marker, string two replaces the ^1 marker, and so on. Thus, we can replace our previous plain vanilla alert with one that describes the reason for the alert like this:

```
A$ = "The open file:"
B$ = "Fred.file"
C$ = "is missing a record! "
D$ = "Would you like to look for it?"
CALL PARAMTEXT (A$, B$, C$, D$)
item = FN STOPALERT (3, 0)
```

Which would appear as shown in Figure 75.

TABLE 12. Default ALRT resources.

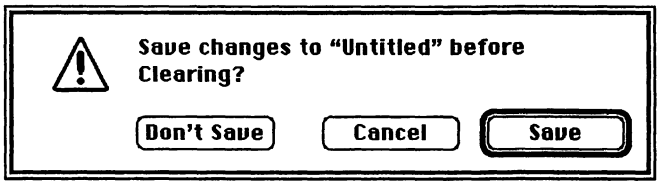
Default alerts:	Explanation
	ALRT #1 — this basic alert only seeks acknowledgment of a user request.
	ALRT #2 —this alert allows the user to cancel the requested action.
	ALRT #3 —This alert has the same setup as ALRT #2 but the button names may be more suitable for some situations.
	ALRT #4 — a modified ALRT #3 where the default action is to cancel the user request.

There are a couple of other default specialty ALRT resources that you can use.

The Save Alert

Another alert included with the runtime package is the save dialog as shown in Figure 75. If the user has changed the information in a document and not saved the changes, use the save dialog when the user tries to close the document or quit the application. It asks the user if they want to **Save**, **Cancel**, or **Don't Save** the changed document.

FIGURE 75. Standard save alert.



The buttons return the following values: **Save** (1), **Don't Save** (2) and **Cancel** (3) so your program can respond appropriately whichever one is chosen.

The General Alert

The general alert provides an quick way of presenting general messages to the user without designing a special ready-made alert. You can pass it one string of information using the PARAMTEXT call. This makes it a great debugging tool for presenting alert messages during program development. Also, this a plain alert, so jazz it up using NOTE-, CAUTION-, or STOPALERT icons when calling it. The code to implement the alert in Figure 75 is shown in Program 116.

PROGRAM 116. General alert procedure.

```
tmp$ = "An example alert message that tells the user something important."
CALL PARAMTEXT (tmp$, "", "", "")
item = FN NOTEALERT (132, 0)
```

FIGURE 76. Standard generic alert.

Creating an ALRT Resource

Here are the steps used to create a custom alert. Open *ResEdit* and then open the *SimpleBase.rsrc* file we created in the last chapter. Select **Create New Resource** from the **Resources** menu. When the resource type window appears, select ALRT in the scrolling list or enter ALRT into the editable field. Click **OK**. *ResEdit* creates a new blank ALRT resource in the *SimpleBase.rsrc* file and opens the ALRT/DLOG editor window shown in Figure 77.

The first thing to do is resize the ALRT using the small black handle in the mini-screen display, or just enter a size in the fields provided. After that, open the DITL editor by double-clicking on the ALRT window in the example screen to start adding buttons and other items.

The DITL editor provides a floating palette containing drag-off objects that can be used in an ALRT or DLOG resource. See Figure 78 for a brief look of the palette tools.

As soon as you are done positioning all of the objects in the DITL editor, close the editor windows and save your work. Later in the chapter we'll see how we create several alerts for our program.

You can modify an ALRTs behavior as an alert box. When it appears, which button is the default one, and when will it beep. All of that is taken care of in the **Set 'ALRT' Stage Info** dialog under the **ALRT** menu item. Figure 79 shows the 'ALRT' Stages dialog with all of its options.

FIGURE 77. ALRT/ DLOG editor window.

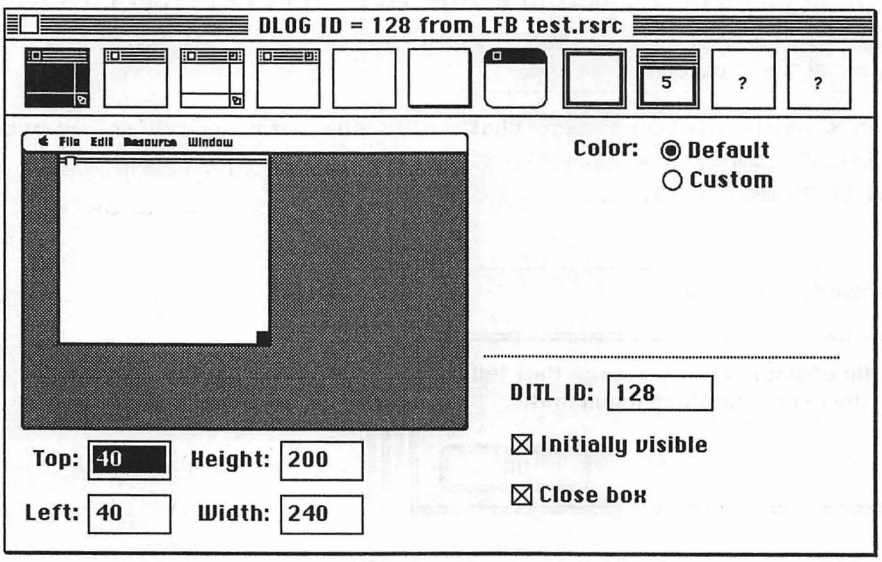
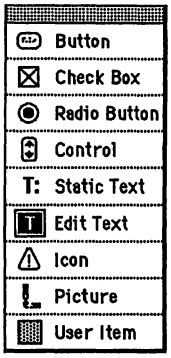


FIGURE 78. DITL floating palette tools.

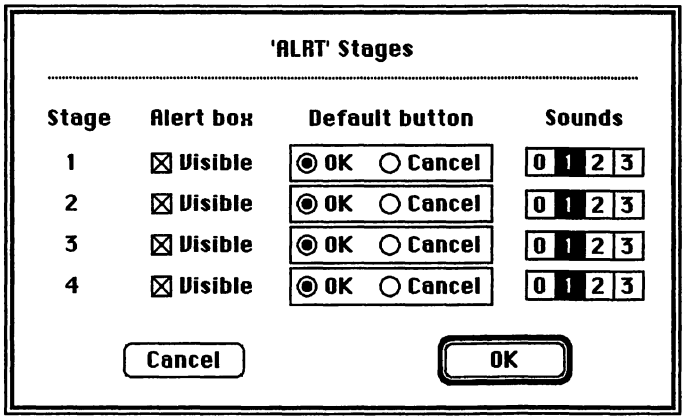


Push button
Checkbox button
Radio button
Custom control
Static text box
Editable text box
Icon image (ICON)
Picture image (PICT)
Custom user item

Click and drag an object tool off the floating palette onto the DITL editor window.

Double click on any object in the DITL editor to view and set that object's information.

FIGURE 79. Set 'ALRT' Stage info dialog.



Regular Exercise

Now that we understand alerts better, let's add a few to *SimpleBase*.

Alert Constants

There are three alerts we will use in *SimpleBase*: the about, field data too long, and no match found. The constants to define them are shown below:

```
_aboutALRT = 128
_twoLongALRT = 129
_noFindALRT = 130
```

About Alert

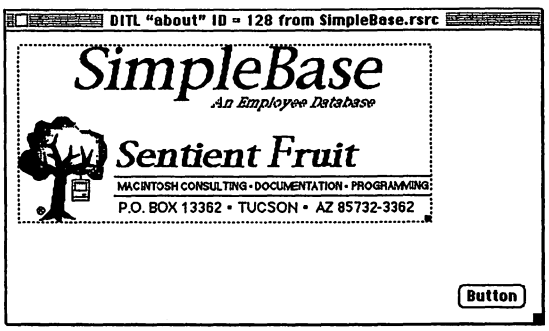
We've mentioned the About window many times, but so far haven't done a thing with it. Well, now's the time. Here we'll create an alert window that will be used as our About window. It will viewable by the user when they select **About SimpleBase...** from the menu.

For *SimpleBase*, our About window will consist of a plain alert window that contains two items: a visible picture (from a PICT resource), and an invisible button positioned outside the boundary of the alert.

To start, create a picture in a graphics program, select it with the marquee tool and copy it to the clipboard. Open the *SimpleBase.rsrc* file with *ResEdit*. Choose **Paste** from the **Edit** menu. The graphic will be pasted into the file as a PICT resource. If it's the first one in the file (it should be at this point), it will have a resource ID of 128.

Following the steps described earlier, create an alert. Stretch its size out a bit larger than what's needed, then double-click on the alert window to open the DITL editor. From the floating palette, drag a push button onto the alert. Position it well down in the lower-right corner of the window as shown in Figure 80. This is the hidden button. Don't worry about naming it since the user will never see it anyway.

FIGURE 80. Hidden button in About.



Next, drag the PICT object into the alert. Position it near the top-left of the window. Double-click on it to open the object editor. Enter the PICT ID in the **Resource ID** text field and set the **Enabled** checkbox, then close the object editor. You should see your about picture scrunched into the picture object. Make sure the PICT object is selected and choose **Use Item's Rectangle** from the **DITL** menu. Reposition for the best appearance and close the DITL editor.

Resize the alert in the mini-screen until the push button is completely hidden and the alert is neatly centered in the window. Then from the **ALRT** menu, choose **Set 'ALRT' Stage Info**. At the dialog, set all of the alert stages to zero.

Finally, save your work, the About window is now done. Return to FB and open *SimpleBase.Incl*. Find the FN ItemAbout subroutine and enter the code shown in Program 117.

PROGRAM 117. ItemAbout alert code.

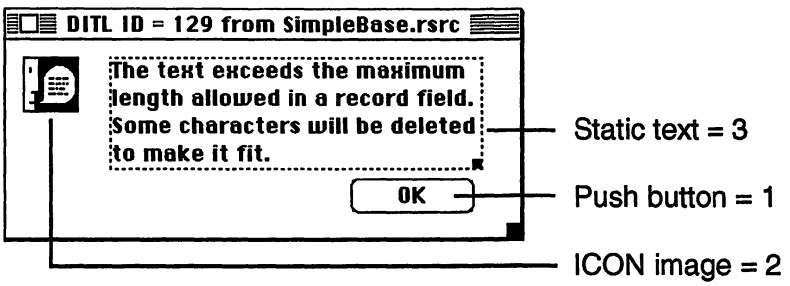
```
LOCAL FN ItemAbout
  item% = FN ALERT (_aboutALRT, 0)
END FN
```

Text Too Long

Create a new ALRT for the "text too long" error. Use the same methods described for the about alert, create an alert that looks like the one shown in Figure 81 with a single **OK** button and a static text field. When done, save your work.

The obvious place to place our "text too long" error message is within the CheckFieldLength$ subroutine. Replace the generic alert code with the custom alert code shown in Program 118.

FIGURE 81. Text too long alert.



PROGRAM 118. Text too long code.

```
LOCAL FN CheckFieldLength$ (fieldID%, maxLen%)
  tmp$ = EDIT$(fieldID%)
  LONG IF LEN (tmp$) > maxLen%
    item% = FN ALERT (_tooLongALRT, 0)
    tmp$  = LEFT$ (tmp$, maxLen%)
  END IF
END FN = tmp$
```

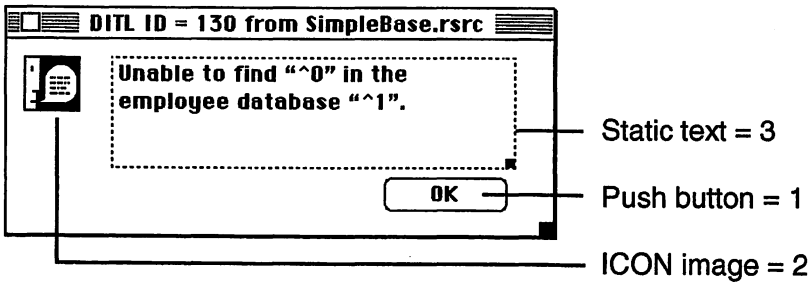
Find Failure

The final alert we need to add informs the user of an unsuccessful search when the Find menu item of button is used. It's a variation on the text too long alert. Since PARAMTEXT allows us to customize our alert messages, we'll beef up the standard "No match found" message using ParamText to add gSearch$ and gFileName$ using a the ^0 and ^1 markers.

To create the alert, duplicate both the ALRT and DITL resources numbered 129. Next open the DITL editor and change the static text item to what is shown in Figure 82. Note the use of the ^0 and ^1 markers to show where the custom text will be inserted. When done, save your changes.

To implement this alert, find the DBFindRecord function and replace the LONG IF/END IF action block with the lines shown in Program 119.

FIGURE 82. Find was unsuccessful alert.



PROGRAM 119. Find unsuccessful alert.

```
LONG IF found = 0
   CALL PARAMTEXT (gSearch$, gFileName$, "", "")
   item% = FN ALERT (_noFindALRT, 0)
   gOpenRecord% = originalRecNum%
   FN DBReadRecord
END IF
```

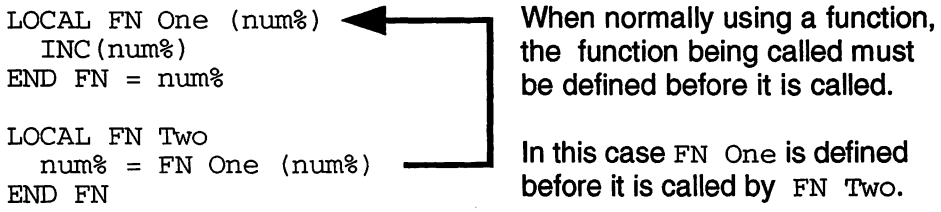
Peak Performance

Closely related to alerts are their big brothers, dialogs. Dialogs are covered very well in the FB *Handbook*, but only on a one-to-one basis. For this dialog you write one subroutine, for that one a different one. Wouldn't it be nice if you could write one dialog handling subroutine and modify it to use different dialogs? Sure it would, and here's how to do it.

A major stumbling block to writing our universal dialog handler is the fact that each dialog has different items associated with it. How can we write one dialog handler that deals with them all? To accomplish that, we're going to re-use `@FN` and `FN USING` to direct program control to the correct subroutines for each dialog. We're also going to create three template routines that our dialog handler needs.

Let's recap how functions work very briefly. The runtime expects you to define a function before it's called in the program. If you call a function that hasn't been defined, an error occurs. In Figure 83 you can see that `FN Two` is defined after `FN One`. Thus `FN Two` can successfully call `FN One`, but `FN One` can't call `FN Two` because of its location in the program.

FIGURE 83. Normal functions in action.



Now, we saw in the chapter "Globals & Includes" how we could use `@FN` to get the address of any function in memory, and we saw how we could use that address with `FN USING` to jump to that subroutine, no matter where it was located in a program. We're going to do the same thing here.

`FN USING` needs two pieces of information. It needs a template definition to tell it what parameters it should accept, and it needs the address of another subroutine. Since the template controls the parameter list, and since a parameter list can't be changed once the template is defined, `FN USING` is stuck with those same parameters. However, it is not stuck with a static subroutine address. Change the address and you can call different routines that accept identical arguments.

FIGURE 84. FN USING in action.

```
LOCAL FN OneTemplate (num%)
END FN = num%

LOCAL FN Two (num%, templatePtr&)
  num% = FN OneTemplate USING templatePtr&; (num%)
END FN

LOCAL FN One (num%)
  INC(num%)
END FN = num%

FN Two (1, @FN One)
```

FN USING lets you call any function based upon its address in memory.

In this case, when FN Two is called it calls FN OneTemplate, but the address in templatePtr& re-directs program control to FN One.

Examine the code in Figure 84, it shows how this works. Any call to FN Two invokes FN USING to call the FN OneTemplate definition. But, the address FN USING calls can be any subroutine in the program, in this case FN One, that accepts the identically assigned parameters. Every time FN Two is called, we send it the address of the subroutine we want FN USING to execute.

To see exactly how this works with some working code, see the example in Program 121. It uses different addresses to enable a single subroutine to return three different results. By passing the address of FN DoAdd, FN DoSubtract, or FN DoMultiply with @FN, FN Math can return a variety of numerical results.

Let's recap, the general requirements of creating subroutines that can handle multiple branching are:

1. A template function to define the parameter list.
2. A FN USING statement that calls the template.
3. A subroutine address that will do the work.

PROGRAM 120. Multiple subroutine program example.

```
LOCAL FN MathTemplate (num%) '<<-- our template subroutine
END FN = num%
LOCAL FN Math (num%, templatePtr&)
   num% = FN MathTemplate USING templatePtr&; (num%)
END FN = num%
LOCAL FN DoAdd (num%)
   INC(num%)
END FN = num%
LOCAL FN DoSubtract (num%)
   DEC(num%)
END FN = num%
LOCAL FN DoMultiply (num%)
   num% = num% * num%
END FN = num%
num% = 2
WINDOW 1
PRINT "Add: ";FN Math (num%, @FN DoAdd)
PRINT "Sub: ";FN Math (num%, @FN DoSubtract)
PRINT "Mul: ";FN Math (num%, @FN DoMultiply)
STOP
```

Handling Multiple Dialogs

Now that we've seen the basic workings, let's apply it to a universal dialog handler. There are the five basic actions required to effectively handle a custom dialog:

1. Get the dialog resource.
2. Do any initialization and setup before showing the dialog.
3. Handle events while the dialog is active.
4. Do any clean-up work when finished.
5. Close the dialog.

We'll start by defining three basic template subroutines which each of our dialogs will need. One is the routine for pre-processing (Step 2), one for event processing (Step 3), and one for post-processing (Step 4) of the dialog. Each requires slightly different parameters and are defined in Program 121.

Each of these template routines tell the runtime the type and number of parameters that should be passed to the working subroutine. For example, `FN PreProcessTemplate` only requires a pointer to the dialog to work, while `FN`

PROGRAM 121. Dialog processing routines.

```
LOCAL FN PreProcessTemplate (dlogPtr&)
END FN
LOCAL FN EvntProcessTemplate (dlogPtr&, itemHit%)
END FN
LOCAL FN PostProcessTemplate (dlogPtr&)
END FN
```

`EvntProcessTemplate` requires both a dialog pointer and the number of the item clicked on in the dialog.

- *As you might have noticed from our earlier example, the pre- and post-processing functions could actually use a single template routine since they both accept a single long integer, but for clarity we'll just define both.*

Once the templates are defined, it's time to write the dialog handling routine that will control multiple dialogs. As you can see in Program 121, the complete `FN HandleModalDialog` is a bit complex but follows our five steps.

Notice the addition of five parameters to the dialog handling function. The `dlogID%` is obvious, it's the resource ID of the `DLOG` resource to open. `itemID%` is the button ID that we'll use as a flag to close the dialog. The next three variables, `preProcessPtr&`, `evntProcessPtr&`, and `postProcessPtr&`, are the addresses to the three processing subroutines that will handle the dialog.

The subroutine opens the `DLOG` resource specified by `dlogID%`, then checks to ensure we have a valid pointer and sets our port to the dialog itself. Next, it uses `FNUSING` to call the specified pre-processing subroutine. Notice the `LONG IF` check to ensure that we have a valid subroutine address. Some dialogs may not require any pre-processing, so this simple check allows us to skip `FN USING` if it's not needed.

Next, we enter a standard `DO/UNTIL` loop with `MODALDIALOG` inside. Here we also check for a valid event-processing pointer before calling that subroutine. When `itemHit%` is less than or equals `itemID%`, control drops out of the loop and any post-processing is taken care of before closing the dialog and resetting the original port.

The final item to take care of is to actually write the subroutines used by each individual dialogs for processing. Since each dialog will have its own unique requirements, you'll have to determine which ones are needed, and write them yourself.

PROGRAM 122. Multiple dialog handling function.

```
CLEAR LOCAL
LOCAL FN HandleModalDialog (dlogID%, itemID%, preProcessPtr&,¬
                  evntProcessPtr&, postProcessPtr&)
  dlogPtr& = FN GETNEWDIALOG (dlogID%, 0, _zTrue)
  LONG IF dlogPtr&
    CALL GETPORT (originalPort&)
    CALL SETPORT (dlogPtr&)
    LONG IF preProcessPtr&
      FN PreProcessTemplate USING preProcessPtr&;(dlogPtr&)
    END IF
    DO
      CALL MODALDIALOG (0, itemHit%)
      LONG IF evntProcessPtr&
        itemHit% = FN EvntProcessTemplate USING evntProcessPtr&; ¬
                  (dlogPtr&, itemHit%)
      END IF
    UNTIL itemHit% <= itemID%
    LONG IF postProcessPtr&
      FN PostProcessTemplate USING postProcessPtr&;(dlogPtr&)
    END IF
    CALL CLOSEDIALOG(dlogPtr&)
    CALL SETPORT (originalPort&)
  END IF
END FN
```

Cooldown

In this chapter we've discussed what alerts are, examined the various types, and seen how to use several standard ones included with every application. We also took the opportunity to add several important alert notices to *SimpleBase* so that the user would be kept informed of program activity.

Strings & Text

Warm-up

The information presented in this chapter provides all of the tools you need to become a text importing and exporting expert. In this chapter you will learn how to:

- Read and save STR resources,
- Read and save STR# resources,
- Read and save TEXT and ZTXT resources, and
- Move INDEX$ arrays to and from resources and edit fields.

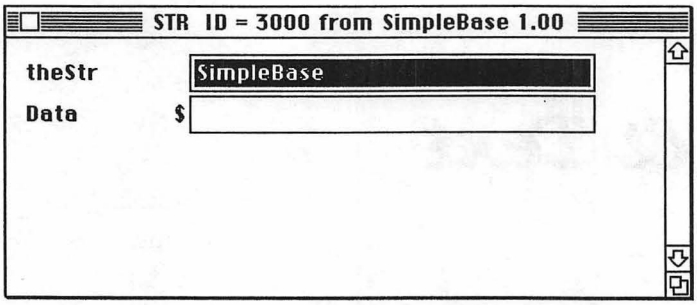
STR Resources

A 'STR ' resource (don't forget the space) contains a single Pascal formatted string with up to 255 characters. STR resources are useful for individual string items unrelated to others in a program.

Creating STR Resources

You create a STR resource by choosing **Create New Resource** from the **Resource** menu and selecting STR from the list of available resource types. Click the **OK** button and a new STR resource is created. Once it's created, use the STR editor shown in Figure 85 to enter up to 255 characters into the string.

FIGURE 85. Editing a STR resource.



Reading a STR Resource

Reading a STR resource requires a few Toolbox Resource Manager calls to open and manipulate the resource. These are shown in Program 123 as a function designed specifically to read individual STR resources.

The subroutine starts by getting the specified string resource using GetString and storing it into a handle in memory. As always, check for a valid handle and be sure that no error occurred during the resource operation before continuing. Any error at this point sounds a double beep and an error message is returned by the function.

If a valid handle is present and no error discovered, it uses PEEK LONG[1] to dereference the handle into a pointer, then PSTR$ to read the resource string into a string variable. Finally, we free the resource from occupying memory with the Toolbox procedure ReleaseResource.

PROGRAM 123. Reading a STR resource.

```
LOCAL
DIM tmp$
LOCAL FN ReadSTRResource$ (resID%)
  strH& = FN GETSTRING (resID%)
  LONG IF (strH& = 0) OR (FN RESERROR <> _noErr)
    BEEP : BEEP
    tmp$ = "Didn't find requested string."
  XELSE
    tmp$ = PSTR$([strH&])
    CALL RELEASERESOURCE (strH&)
  END IF
END FN = tmp$
```

1. Note that we use the shorthand method of PEEK LONG by using square brackets to read the handle.

Saving a STR Resource

To save a STR resource, it's a matter of going backwards from what was done to get the string. The example in Program 124 demonstrates how to add or replace a STR resource quickly and easily.

It starts by passing a string variable and a resource ID to the SaveSTR-Resource function. It then uses the Toolbox function NewString to create a handle from the string variable. If the handle is valid, the subroutine next checks for the presence of an older copy using the same resource ID in the resource fork. If one is found, it's must be removed from the file using FN RmveResource before saving the new one.

A call to the AddResource procedure adds the string handle to the resource file. Next, we inform the Memory Manager that we've modified its resource map with ChangedResource (so that our changes will be saved to disk), and finally use ReleaseResource to free the string handle memory.

As you may have noticed, reading or adding STR resources requires some work with Toolbox routines. While that's true when dealing with STR resources, accessing other string and text resources is more directly supported in the next section.

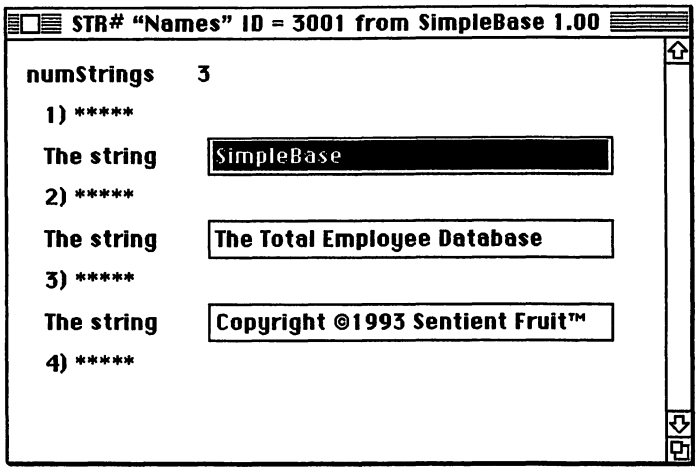
PROGRAM 124. Saving STR resources.

```
LOCAL FN SaveSTRResource (tmp$, resID%)
  strH& = FN NEWSTRING (tmp$)
  LONG IF (strH& = 0) OR (FN RESERROR <> _noErr)
    BEEP : BEEP
    tmp$ = "Can't create string handle."
  XELSE
    tempH& = FN GETSTRING (resID%)
    IF tempH& THEN CALL RMVERESOURCE (tempH&)
    CALL ADDRESOURCE (strH&, _"STR ", resID%, "")
    CALL CHANGEDRESOURCE (strH&)
    CALL RELEASERESOURCE (strH&)
  END IF
END FN
```

STR# Resources

A STR# resource (pronounced string list) contains a list of Pascal formatted strings. STR#s are used so often in programming the Macintosh that allowing easy access to them with an FB function was absolutely necessary. In this case, it's a function called STR# (not to be confused with STR#, the resource type).

FIGURE 86. Editing a STR# resource.



Using STR# resources has several benefits. First, memory isn't used by STR# the same way DIM'ed string arrays do. The STR# list resides on disk until needed, then is loaded into memory only as long as it takes to read an individual string, then it's released. This is unlike regular string arrays which always occupy memory. Second, you can have hundreds of STR# resources in a program's resource fork, accessible and easy to use.

If you are only using a few strings, saving them to a STR# resource may be overkill. However, I once saw a program that used a single dimensioned string array to store over 600K of strings. Half of the program was used to assign the strings to elements of the string array. I recommended converting the string array into several STR# resources, with the result that the program ran faster, the source code was reduced in size by nearly half, and the final application operated in less than half its previously allocated memory space. There are big benefits to using STR# resources.

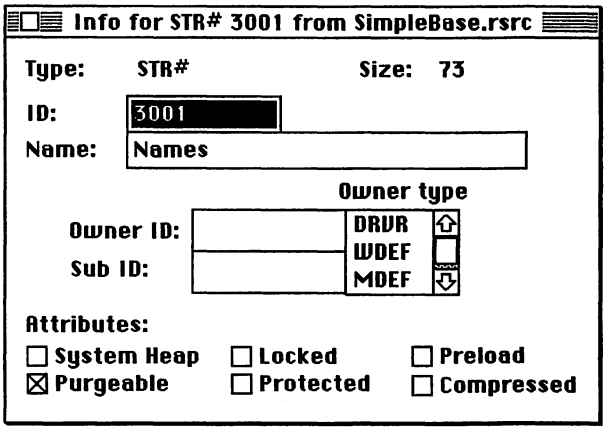
Creating STR# Resources

Creating STR# resources is a snap. *ResEdit* contains a STR# editor that creates STR# resources quickly and easily. Open your resource file and choose **Create New Resource** from the **Resources** menu, choose STR# from the list, then click **OK**. Once the STR# resource is created, use the STR# editor shown in Figure 87 to enter the string data.

Inserting Fields & Deleting STR# Resources

To add a string to a STR# resource, click on the five asterisks in the STR# editor and select **Insert New Field(s)** from the **Resources** menu. *ResEdit*

FIGURE 87. STR# resource information.



will insert a blank string into the position indicated by the asterisks. To delete an individual string, choose the asterisks next to the string to delete, then choose **Clear** from the **Edit** menu. Examine Figure 87 again to see how we added three strings to a STR# resource.

Setting STR# Resource Info

Use **Get Resource Info** from the **Resource** menu to change the STR# resource ID, add a name, and change attributes as shown in Figure 87. At a minimum, always set the **Purgeable** checkbox to ensure that the STR# resources will never occupy memory longer than necessary.

Reading STR#

To read a STR# resource use *FB's* STR# function. It only requires two parameters, the STR# resource ID and the indexed ID of the string to read. For example, to read the second string in STR# 3001 as shown in Figure 87, do this:

```
PRINT STR# (3001, 2)
```

Which will output:

```
The Total Employee Database
```

Occasionally, you may need to determine how many strings are actually stored in a STR# either for a loop block or program use. This information is stored in the first two bytes of all STR# resources. The function shown in Program 125 shows how to get this information from a specified STR# resource.

PROGRAM 125. Getting a string count from a STR# resource.

```
LOCAL FN GetSTRCount% (resID%)
  strCount% = 0
  resH& = FN GETRESOURCE (_"STR#", resID%)
  LONG IF (resH& = 0) OR (FN RESERROR <> _noErr)
    BEEP
  XELSE
    strCount% = resH&..none%
    CALL RELEASERESOURCE (resH&)
  END IF
END FN = strCount%
```

Saving STR#

You can save string data to a STR# resource using DEF APNDSTR. If the resource is already present do this:

```
strH& = FN GETRESOURCE (_"STR#", resID%)
LONG IF strH&
  DEF APNDSTR (stringVar$, strH&)
  CALL CHANGEDRESOURCE (strH&)
END IF
```

If the required STR# resource is not present, then create it using the CreateSTRResource function shown in Program 126.

PROGRAM 126. Creating a STR# resource.

```
LOCAL FN CreateSTRResource& (resID%)
  strH& = FN NEWHANDLE (2)
  LONG IF strH& = 0
    BEEP : BEEP
  XELSE
    CALL ADDRESOURCE (strH&, _"STR#", resID%, "")
    CALL CHANGEDRESOURCE (strH&)
  END IF
END FN = strH&
```

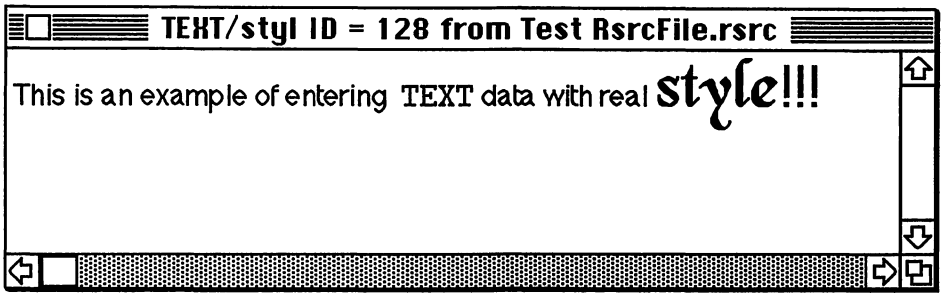
TEXT Resources

A TEXT resource can contain up to 32,767 characters in a single resource. The characters in a TEXT resource can also be styled using the styl resource.

Creating TEXT Resources

Just as we've done before, choose **Create New Resource**, select TEXT as the resource type, and then begin entering your data in the new resource. You

FIGURE 88. Editing TEXT and styl resources.



can use the **Font**, **Size**, and **Style** menus that appear on the menu bar to specify the formatting of your text data. When any of these three menus are used, a styl resource of the same resource ID is added to the file. The TEXT and styl editor is shown in Figure 87.

Reading TEXT Resources

Reading a TEXT resource into an edit field is simple. Just give either EDIT$ or EDIT FIELD a valid resID% and let the runtime do all of the work. The example in Program 127 shows exactly how to do this using a resource ID. Remember that if you plan to read a styled TEXT resource, you must have a styled edit field to receive it.

PROGRAM 127. Reading TEXT data.

```
LOCAL FN ReadTEXTResByID (resID%, fieldID%)
  EDIT$ (fieldID%) = %resID%
END FN
```

There is a trick to reading a TEXT resource into a handle and placing it into an edit field. The trick is that the handle must be configured as a ZTXT resource in order to properly read into an edit field. All this involves is the addition of a length value at the beginning of the handle. Let's see how to do that.

Converting TEXT to ZTXT

A ZTXT resource contains up to 32,767 characters and appended style data associated with the text. ZTXT is a styled text format returned by the GET FIELD function. It combines an edit field's normal text with its style information into a single handle that can be written to disk or resource.

Converting a normal TEXT resource into ZTXT format is accomplished as shown in Program 128. The procedure is to create a handle 2 bytes larger

PROGRAM 128. Converting TEXT to ZTXT format.

```
LOCAL FN PutTextH2Field (textH&, fieldID%)
  EDIT$ (fieldID%) = &textH&
END FN
LOCAL FN Text2Ztxt& (textH&)
  ztxtH& = 0
  LONG IF textH&
    textHSize% = FN GETHANDLESIZE (textH&)
    ztxtH& = FN NEWHANDLE (textHSize% + 2)
    LONG IF (ztxtH& = 0) OR (SYSERROR <> _noErr)
      BEEP
    XELSE
      osErr% = FN HLOCK (textH&)
      osErr% = FN HLOCK (ztxtH&)
      BLOCKMOVE [textH&], [ztxtH&] + 2, textHSize%
      ztxtH&..none% = textHSize%
      osErr% = FN HUNLOCK (ztxtH&)
      osErr% = FN HUNLOCK (textH&)
      DEF DISPOSEH (textH&)
    END IF
  END IF
END FN = ztxtH&
```

than the TEXT resource, then copy the TEXT data into the ZTXT handle with BLOCKMOVE but offset from the start of the handle by 2 bytes. Next, poke the text length into the first 2 bytes of the ZTXT handle. The text is now formatted correctly for insertion into an edit field.

Note that we have not included routines for adding the styl resource. Routines for doing exactly that can be found in the *Functions Library Help* file on your FB disks.

Saving ZTXT Resources

Saving a ZTXT handle is much easier than converting one. The procedure uses GET FIELD to gather the text and style data from an edit field into a single handle. Next, write the handle to disk with Toolbox calls exactly as was shown with the STR resources. Additionally, ensure that we don't duplicate a ZTXT resource by deleting any older version before writing the new one. The whole procedure is shown in Program 129 in the function SaveZTxt-2Rsrc.

PROGRAM 129. Saving ZTXT data to a resource.

```
LOCAL FN SaveZTxt2Rsrc (resID%, fieldID%)
  GET FIELD zTxtH&, fieldID%
  LONG IF zTxtH& = 0
    BEEP : BEEP
  XELSE
    tempH& = FN GET1RESOURCE (_"ZTXT", resID%)
    IF tempH& THEN CALL RMVERESOURCE (tempH&)
    CALL ADDRESOURCE (zTxtH&, _"ZTXT", resID%, "")
    CALL CHANGEDRESOURCE (zTxtH&)
    CALL RELEASERESOURCE (zTxtH&)
    KILL FIELD zTxtH&
  END IF
END FN
```

Reading ZTXT Resources

First, read the ZTXT resource into a handle with the Toolbox function Get1Resource. If it's a valid handle, use the EDIT$ function to replace the edit field's current contents. Since it's already been formatted correctly, we can insert the ZTXT handle into a styled edit field. Finally, clear the old resource handle from memory with ReleaseResource. An example of doing exactly this is shown in Program 130.

PROGRAM 130. Reading ZTXT data.

```
LOCAL FN ReadZTxt2Rsrc (resID%, fieldID%)
  zTxtH& = FN GET1RESOURCE (_"ZTXT", resID%)
  LONG IF (zTxtH& = 0) OR (FN RESERROR <> _noErr)
    BEEP : BEEP
  XELSE
    EDIT$ (fieldID%) = &zTxtH&
    CALL RELEASERESOURCE (zTxtH&)
  END IF
END FN
```

Regular Exercise

With our new knowledge of various types of text and strings resources available, let's use that knowledge to add some to our program.

Creating Program STR#

Actually, the best method of creating STR# resources is not to use *ResEdit*. Instead, make use of the COMPILE statement setting _strResource. This tells the compiler that all strings it encounters in a program should be added to the final application as a single STR# (ID 127) when you build the program.

- *Note that you must use the _strResource setting if a program makes any use of INCLUDE files.*

Field & Button STR#

Using a STR# list is a great way of making your application easy to localize for a foreign country. While most of us have no intention of marketing our software products overseas, some people do, and STR# makes translating programs easy without re-writing the code. It's for this reason that you should use STR# resources to hold all the window, button, and static field text in a program.

Open *SimpleBase.glbl* and add the following constants:

```
_windowSTR = 1000
_buttonSTR = 2000
_fieldSTR = 3000
```

Save your changes. These few constants enable us to determine the STR# ID used for storing the text. All window titles will be found in the STR# number 1000, which can have up to 63 window titles in it. Button titles will be found in STR# ranging from 2001 through 2063, while field text ranges from 3001 to 3063. Let's see how to use this.

Window STR#

Open *SimpleBase.main* and rewrite all of the window build routines to take advantage of STR#. Remember how we used a tmp$ variable to assign names to windows, buttons, and static fields. Convert all of those to the STR# command. For example, in FN BuildEntryWindow, replace:

```
tmp$ = "SimpleBase Data Entry"
```

with:

```
tmp$ = STR# (_windowSTR, _dbEntryWIND)
```

Note how we can use the _windowSTR constant to open the correct STR# resource, and the title is found using the constant assigned to the window.

Button & Field STR#

For all of the buttons in the window, combine the constant for button STR# with the window constant like this, using the btnID% as the offset into the STR# entries:

```
tmp$ = STR# (_dbEntryWIND_buttonSTR, btnID%)
```

And do the same for the window's static text fields:

```
tmp$ = STR# (_dbEntryWIND_fieldSTR, btnID%)
```

This identification method makes it easy to read your source code and see which STR# resource is accessed. It also makes it easy to change a string after the program has been built. Forget re-compiling, just edit the STR# resources in *ResEdit* and away it goes.

- *Notice how we combine the window and string list constants. The runtime determines the calculated value upon compile time and inserts the correct value into the code when it built. If we had used a plus sign between the two, the runtime would have to calculate the value each time the line was called. Doing it the first way is much faster.*

Make the code changes on all of the window build routines and save them to disk. Then use *ResEdit* to create the STR# resources and the strings themselves for the subroutines to call.

Creating the Help TEXT

With *SimpleBase.rsrc* open in *ResEdit*, create a new TEXT resource using the **Create New Resource** item on the **Resources** menu, enter TEXT, and click **OK**. Select it, then choose **Get Resource Info** and make it **Purgeable**. Close and save when done. Finally, duplicate the TEXT resource nine times and renumber them from 1001 to 1009.

With the TEXT resources created, add the information for all four menus and each window as I did. Examine the *SimpleBase.rsrc* file to see the text I used to describe the various menu items in their respective help text resource.

Next, open the *SimpleBase.glbl* file and add these constants:

```
_minHelpID = 1001
_maxHelpID = 1009
```

Save your changes and open the *Simplebase.main* file. Go to the FN DialogHelpWindow function and modify it to look like the example shown in Program 131. Save your changes and test the Help window's information, as well as try out the About and string length error alerts.

PROGRAM 131. Help window dialog handling.

```
LOCAL FN DialogHelpWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    CASE _btnClick
      LONG IF dlgID% > _helpSCROLL
        LONG IF dlgID% = _prevHelpBTN
          DEC (gHelpID%)
          IF gHelpID% < _minHelpID THEN gHelpID% = _maxHelpID
        XELSE
          INC (gHelpID%)
          IF gHelpID% > _maxHelpID THEN gHelpID% = _minHelpID
        END IF
        SCROLL BUTTON _helpSCROLL, 1
        EDIT$(_helpSCROLL) = %gHelpID%
      END IF
    CASE ELSE
  END SELECT
END FN
```

Peak Performance

There is one other form of managing strings that we haven't talked about, and that is INDEX$ arrays. A very powerful set of commands enables you to easily manipulate these very flexible string arrays. The commands include INDEX$, INDEX$ I, INDEX$ D, INDEX$ F, CLEAR and the MEM function.

INDEX$ Types

An INDEX$ array is only limited by the amount of memory available to the program. The array itself can contain millions of elements (memory permitting of course) with each element able to store a string with up to 255 characters. A program can have a total of ten INDEX$ arrays open at one time and all INDEX$ arrays are global in nature. This means that all LOCAL FNs see and use them with no additional work on the part of the programmer.

- *A small secret, an INDEX$ array is really a handle in disguise. That's why it can grow and shrink as demands upon it change.*

INDEX$ arrays come in either variable or fixed length elements. The variable length format enables the array to have elements that range between 0 to 255 characters. Since each element can be a different length, searches, insertions, and deletions are somewhat slower since each element must be individually examined by each command. A fixed length array has a defined

maximum length for each element. This limits the type of data stored in the array but has a much faster search, insertion and deletion speed than a variable length array does.

- *There's a correction to the Reference manual. When defining a fixed length INDEX$ array, the `bytes&` variable should be replaced by the `numElems&` variable.*

Creating an INDEX$ Array

To create an INDEX$ array you need a location in memory to contain the data that will be stored there. To do that, use the CLEAR statement. For example, to create a small variable length INDEX$ array do this:

```
CLEAR numBytes&, indxID%
```

Where indxID% represents a number between zero and nine. Each array can have its own memory requirements. To create a fixed length INDEX$ array do this:

```
CLEAR numElems&, indxID%, elemLength%
```

Once the array is created, you may sometimes need to clear out old data and insert new. To clear an INDEX$ array simply do this:

```
CLEAR INDEX$ indxID%
```

where again, indxID% represents the INDEX$ array to clear.

INDEX$ Information

Use the MEM function to return information about a specific INDEX$ array. You can find out how many elements exist in an INDEX$ array, how much memory it currently uses, how much memory it has free and more. For example, to determine how many entries there are in a particular INDEX$ array do this:

```
numElements% = MEM (indxID% + _numElem)
```

To determine how much memory is left for additional elements use:

```
bytesAvailToUse% = MEM (indxID% + _availBytes)
```

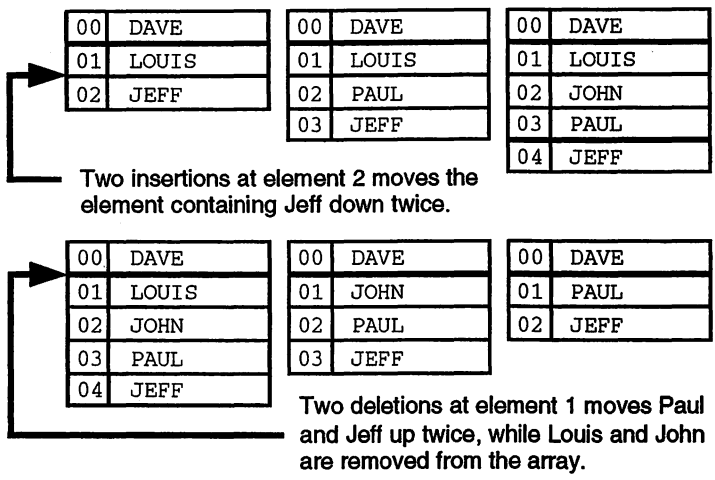
And to get the maximum number of bytes available use:

```
maxBytesAvail% = MEM (indxID% + _maxBytes)
```

Inserting and Deleting Elements

Once an INDEX$ array is available, you can insert strings into the array using INDEX$ I. INDEX$ I enables you to insert a string into any array element directly.

FIGURE 89. Inserting & Deleting elements.



Unlike a traditional array created with the DIM statement, INDEX$ I shifts the data in the array down one element to make room for the insertion.When it comes time to delete an element using INDEX$ D, the specified element is removed and all subsequent elements are shifted forward by one. The diagram in Figure 89 shows how elements are added and removed to the array.

Changing INDEX$ Sizes

After inserting elements, it's often necessary to increase the size of a particular INDEX$ array to make room for more data. To do that you use CLEAR to resize the specified INDEX$. The subroutine shown in Program 132 shows how to correctly adjust any INDEX$ to accommodate more or less array elements.

PROGRAM 132. Changing an INDEX$ size.

```
CLEAR LOCAL
LOCAL FN SetIndexSize (sizeReq&, indxID%, setSpare%)
  indxSize& = MEM(indxID% + _availBytes)
  crntSize& = MEM(indxID% + _maxBytes)
  LONG IF indxSize& < sizeReq&
    CLEAR crntSize& + sizeReq& + setSpare%, indxID%
    LONG IF MEM(indxID% + _maxBytes) = crntSize&
      BEEP : SYSERROR = _dsMemFullErr
    END IF
  END IF
END FN
```

Displaying INDEX$

People have discovered the many uses for scrolling edit fields, and lots of you are wondering how to mix a scrolling edit field with an INDEX$ array. The following routines show how to move data from an INDEX$ to an edit field.

We start by examining the differences between the two. An INDEX$ array exists as a series of strings separated by a length byte. The INDEX$ commands know how to traverse this format to insert, read, or delete any element in this mass of data. Edit field text, however, is contained in a single handle

PROGRAM 133. INDEX$ to an edit field.

```
LOCAL FN Index2ScrollEF (fldID%, indxID%)
  tmp$      = ""
  strPtr&   = @tmp$ + 1
  count%    = 0
  offset&   = 2
  numElems% = MEM (indxID% + _numElem)
  indxSize& = MEM (indxID% + _usedBytes)
  LONG IF size& > _maxInt
    BEEP : BEEP
  XELSE
    indxH& = FN NEWHANDLE (indxSize& + 2)
    LONG IF (indxH& = 0) OR (SYSERROR <> _noErr)
      BEEP : BEEP
    XELSE
      osErr% = FN HLOCK (indxH&)
      LONG IF osErr% = _noErr
        indxPtr& = [indxH&]
        DO
        tmp$ = INDEX$ (count%, indxID%) + CHR$(13)  'add linefeedt
          newSize& = LEN (tmp$)
          BLOCKMOVE strPtr&, indxPtr& + offset&, newSize&
          offset& = offset& + newSize&
          INC (count%)
        UNTIL count% = numElems%
        osErr% = FN HUNLOCK (indxH&)
        osErr% = FN SETHANDLESIZE (indxH&, offset&)
        indxH&..none% = offset& - 2
      END IF
      EDIT$ (fldID%) = &indxH&
      SCROLL BUTTON fldID%,1,1, numElems%
    END IF
  END IF
END FN
```

containing all the text and formatting characters (like line feeds), but not style information.

Thus, to mix the two, it's necessary to translate one format (separate elements) into another (single handle of data). The function in Program 133 does all the necessary translation, moving the single elements of an INDEX$ array, into a single ZTXT handle for insertion into the specified edit field. It creates a handle large enough to hold the INDEX$ data, then inserts each element into the handle while it appends a line feed at the end of each insertion. When done, it resizes the handle to reflect the actual character count, places the count in the first two bytes of the handle, then uses EDIT$ to place the handle into the designated field.

Cooldown

Once again we've covered a lot of ground. In this chapter we described the three types of string and text resources including STR, STR#, and TEXT. We also saw how to read and save data to them. And covered INDEX$ arrays and how to manage strings in them.

CHAPTER 16

Edit Menus

Warm-up

This chapter describes how you can write programs that implement an Edit menu and communicate with other programs via the Clipboard. In this chapter we will learn:

- How to use the standard Edit menu,
- How to customize the Edit menu, and
- How to cut, copy, and paste `TEXT` and `PICT` data.

The Edit Menu

A standard feature of Macintosh programs is the ever present Edit menu. The Edit menu provides common editing capabilities that enable a user to transfer information via cutting or copying, from one position to another in the program, or from program to program. This editing capability is not restricted to text alone. It can consist of pictures, records, or anything else the programmer can codify into a selectable object.

The FB runtime handles the text handling capabilities for you when dealing with text information in edit fields. It can't, however, handle graphics, records, or anything else automatically. You must program that capability into it yourself. And that's just what we're going to do.

The Standard Edit Menu

To use the standard Edit menu, use the EDIT MENU statement in your program. FB will create an entire Edit menu containing the **Undo**, a divider, **Cut**, **Copy**, **Paste**, and **Clear** items. Once added to a program, the user can easily manipulate text as it appears in an edit field. It can't deal with text in PRINT or INPUT statements, only text in edit fields.

- *Note that the default Edit menu items are stored in a STR# resource in the FutureBASIC Extras file.*

Undo, however, is not supported by the runtime. That too must be programmed into the application by the programmer.

Clipboard Workings

The Clipboard is referred to in programming as the **desk scrap**. It is the mechanism by which different applications can share data in common formats.

TextEdit has its own internal version of the Clipboard, known as the **TE scrap**. It is the TE scrap that holds text data during cutting and pasting between edit fields by the runtime. However, the TE scrap must be passed to the desk scrap if the application wishes to share TEXT data.

The normal method of transferring the TE scrap to the desk scrap occurs when the application receives notice that it is being moved to the background via a _mfSuspend event. At that time the frontmost application needs to transfer the TE scrap to the desk scrap for possible use by another program.

When the application is brought to the foreground once again, it receives a _mfResume event that tells it to transfer the desk scrap to its own internal TE scrap so the user will have it available.

Fortunately we don't need to worry about either of these events when dealing with TEXT, the runtime manages to take care of these details for us. But, we must worry about them when it comes time to handle PICT data ourselves. We will bypass these problems, however, by transferring the picture data directly to the desk scrap on cut or copy operations, and read the data directly from the desk scrap when it comes time to paste the picture.

Getting Scrap Information

There are two scrap types we can check the Clipboard for using the WINDOW function, they are TEXT and PICT. For example, to determine if a PICT scrap is on the Clipboard do this:

```
pictOnClip% = WINDOW (_pictClip)
```

To check for TEXT, use this:

```
textOnClip% = WINDOW (_textClip)
```

To determine if other scrap types are present, use the Toolbox FN GetScrap. GetScrap requires three parameters, a handle to hold the scrap data, a data type, and an offset variable. When given this information, the function returns the size of the scrap found. If the specified scrap type isn't found, it returns zero. If the scrap type is found, a copy of it is made to the handle passed to it.

If the handle parameter is nil, GetScrap returns the presence of a scrap type. For example, to determine if any scrap on the Clipboard has the type ZTXT do this:

```
scrapSize& = FN GETSCRAP (_nil, _"ZTXT", offset&)
```

We'll see how to get the scrap later on, so let's look at the type of data we need to support.

Customizing the Edit Menu

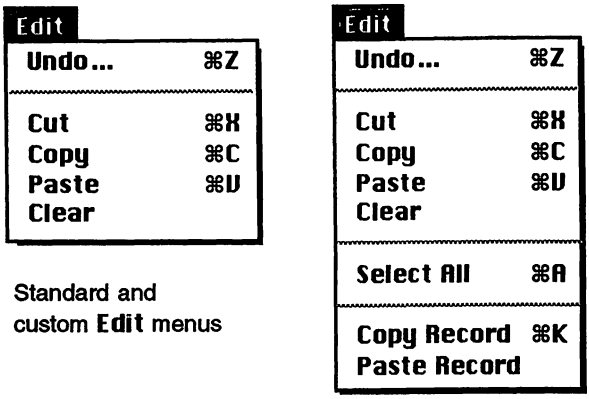
One method of customizing the Edit menu is to add additional items below the standard menu items. For example, when the EDIT MENU statement is used, it automatically includes six items (**Undo**, a divider, **Cut**, **Copy**, **Paste**, and **Clear**). If you assign new items beginning with an itemID% greater than six, the items will be appended to the Edit menu. That's exactly what we've done to append the **Select All**, **Copy Record**, and **Paste Record** items.

Edit Menu Events

The standard Edit menu handling provided by the runtime is just great when it comes to copying and pasting text from edit field to edit field. However, it's not designed to handle anything else. So we have a choice. Do we disable the runtime's standard text editing features and write our own? Or, do we somehow intercept menu choices in the Edit menu and respond accordingly? The answer is neither.

When the runtime sees a window with active edit fields, it's perfectly happy to handle the text editing directly and never let us see a menu event! This works

FIGURE 90. Standard and custom Edit menus.



fine with text, but how can we handle the picture field? Easy, we use an undocumented feature of the runtime. If we disable all of the fields in the Data Entry window with:

```
EDIT FIELD #_nil
```

We are sent Edit menu selections as menu events. Once we receive the menu event, it's not hard to handle the cutting, copying, pasting, or clearing of pictures.

Regular Exercise

Now that we understand what can be done with **Edit** menus, let's begin adding support for them in *SimpleBase*.

To do that we'll create another include file called *EditMenus.Incl*. In it we will place all of our special **Edit** menu routines. We will also add a function to the *Project.Incl* file for the `DoEditMenu` subroutine and a global pointer to the `SimpleBase.glbl` file. Here are the directions to get started:

Create a new source file called *EditMenus.Incl*. Set it up as an include file with the standard program headers used in the other include files. At the very bottom of the new file add:

```
gEditMenuPtr& = @FN pDoEditMenu
```

Then in *SimpleBase.glbl* add:

```
DIM gEditMenuPtr&
```

In *Project.Incl* add:

```
DEF FN DoEditMenu (itemID%) USING gEditMenuPtr&
```

Finally, in the *SimpleBase.Incl* file, delete the `ItemSelectAll`, `ItemExport-Record`, and `ItemImportRecord` subroutines, then cut the `DoEditMenu` subroutine and paste it into *EditMenus.Incl*. Rename it `pDoEditMenu`. Add the subroutines called by `pDoEditMenu` (note the name changes), and remember to save all of your files. The starting source for the *EditMenus.Incl* is shown in Program 135.

Adding Select All

Implementing the **Select All** item is the easiest addition to make. Just add the `SETSELECT` statement to the `EditSelectAll` subroutine as shown in Program 134. Because this is a menu item not handled by the runtime, whenever the user chooses it the contents of the active edit field will be highlighted, ready for cutting, copying, deletion, or replacement.

PROGRAM 134. Select all subroutine.

```
LOCAL FN EditSelectAll
  SETSELECT 0, _maxInt
END FN
```

Exporting Data

All right, we know the runtime will take care of exporting all text selections for us, so all we have to worry about is the picture data. How do we do that? Well, we start by identifying where the user is. Since we are only concerned with the lone picture field in the Data Entry window, let's start there.

The picture field was created as an active field, but the Tab and Shift-Tab events are designed to bypass it when the user presses either combination. The only remaining event to watch for is an `_efClick` in the picture field in the Data Entry window. When *SimpleBase* receives this event, it should deactivate all the active fields in the window so that it can intercept all subsequent menu events.

Also, because the user has no way of knowing that they clicked in the picture field (it doesn't have a blinking cursor), it needs some method of showing them the field is selected. The answer is a routine called `FrameField`. `FrameField` draws a rectangle around the picture field (much like the System 7 get and put field dialogs) so that the user has a visible mark of where they are. It also handles erasing the rectangle when the user clicks in an edit field. The routine is shown in Program 136.

PROGRAM 135. EditMenus.Incl file.

```
' --- HEADER -------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels
' --- CONSTANTS ----------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS
' --- FUNCTIONS ----------------------------------
LOCAL FN EditUndo
END FN
LOCAL FN EditCut
END FN
LOCAL FN EditCopy
END FN
LOCAL FN EditPaste
END FN
LOCAL FN EditClear
END FN
LOCAL FN EditSelectAll
END FN
LOCAL FN EditExportRecord
END FN
LOCAL FN EditImportRecord
END FN
LOCAL FN pDoEditMenu (itemID%)
  SELECT itemID%
    CASE _iUndo      : FN EditUndo
    CASE _iCut       : FN EditCut
    CASE _iCopy      : FN EditCopy
    CASE _iPaste     : FN EditPaste
    CASE _iClear     : FN EditClear
    CASE _iSelectAll  : FN EditSelectAll
    CASE _iCopyRec    : FN EditExportRecord
    CASE _iPasteRec   : FN EditImportRecord
  END SELECT
END FN
' --- GLOBAL POINTER -----------------------------
gEditMenuPtr& = @FN pDoEditMenu
```

PROGRAM 136. FrameField subroutine.

```
LOCAL
DIM rect;8
LOCAL FN DrawFrame (showFrame%)
  CALL SETRECT (rect, 210,113,350,275)
  LONG IF showFrame%
    PEN 2,2,,,0
  XELSE
    PEN 2,2,,,19
  END IF
  CALL FRAMERECT (rect)
  PEN 1,1,,,0
END FN
```

`FrameField` starts by defining a rectangle with enough room to leave a pixel of space around the picture field's boundary. Then, depending on the setting of `showFrame%`, it either draws a black 2 pixel wide rectangle around the field or erases one. When done it resets the pen back to its normal setting. It's called from two locations in `DialogEntryWindow`: `_efClick` and `_wndRefresh`. Both make appropriate checks of the `fieldID%` to properly set the `showFrame%` flag.

The end result of all this is that when the user clicks in the picture field, a border appears around it and all fields in the window are deactivated. All selections in the **Edit** menu, even the ones previously handled by the runtime, are sent to us via a menu event.

Now that we have menu events, let's see how to export pictures.

PROGRAM 137. Modified DialogEntryWindow routines.

```
' ••• WINDOW EVENTS
CASE _wndRefresh
  LONG IF WINDOW(_efNum) = 0
    FN DrawFrame (_true)
  XELSE
    FN DrawFrame (_false)
  END IF
' ••• FIELD EVENTS
CASE _efClick
  FN EFClickEvent (dlgID%)
  LONG IF dlgID% = _dbPhotoFLD
    EDIT FIELD #_nil
    FN DrawFrame (_true)
  XELSE
    FN DrawFrame (_false)
  END IF
```

Exporting PICT

Exporting picture data is bit tricky – we have to deal with a couple of different formats that the picture data may be stored in the picture field. Let's review the rudiments of picture fields before examining the routines required to manipulate pictures.

When a PICTURE FIELD is created, it's possible to add a picture using a either a PICT resource ID, or a picture handle. The runtime stores this picture information in the space normally used to store text data. We need to extract this information and get a copy of the picture.

We also have one other problem, how do we identify a PICTURE FIELD from an EDIT FIELD? The same way the runtime does, by examining the field type. You see, a picture field is just a modified edit field that can show pictures. A field created with PICTURE FIELD always has a negative field type value. We can check this using:

```
fieldType% = WINDOW (_efClass)
```

And if it's less than zero, it's a picture field.

Once a field is identified as a picture field, we have two different methods of storing picture information. In practice, when you pass a picture ID or picture handle to a picture field it stores the ID or handle in the field as well as the method used to place it there. We extract this information using:

```
pictInfo$ = EDIT$(fieldID%)
```

If the first character in pictInfo$ starts with the "%" character, a picture ID follows. If the first character is a "&", a picture handle follows.

PROGRAM 138. Exporting PICT.

```
CLEAR LOCAL
LOCAL FN GetPICTHandle&
  tmp$ = EDIT$ (_dbPhotoFLD)
  pict$ = RIGHT$ (tmp$, LEN (tmp$) - 1)
  SELECT LEFT$ (tmp$, 1)
    CASE "%"
      pictH& = FN GETPICTURE (CVI(pict$))
    CASE "&"
      pictH& = CVI(pict$)
    CASE ELSE
      pictH& = _nil
  END SELECT
END FN = pictH&
```

Our routine to determine all of this is FN GetPICTHandle& and is shown in Program 138. It starts by extracting the text from the picture field using EDIT$. It extracts the picture number from the field using RIGHT$. Next, it determines how the picture data got into the picture field, by resource ID or handle. If the data was placed using a resource ID, we convert the remaining text into a number and use FN GetPicture to get a handle to the picture. However, if the picture was placed as a handle, we convert the remaining text into a handle using CVI. If neither character is found, it sets the handle to nil and the routine returns nothing.

Copy Picture

When the user chooses **Copy** from the **Edit** menu with all fields deactivated, we are sent a menu event. The HandleMenuEvent subroutine routes control to pDoEditMenu. It in turn calls the EditCopy function shown in Program 139. We cover this subroutine first because it is also called by EditCut.

It begins by calling GetPICTHandle& to see if the picture field contains a picture. If it does, it then calls FN DataHandleToScrap to place the picture onto the desk scrap. This makes it available to any other program that reads the desk scrap. Since the Scrap manager makes a copy of our picture handle, it disposes of pictH& with DEF DISPOSEH and ends.

PROGRAM 139. EditCopy subroutine.

```
CLEAR LOCAL
LOCAL FN EditCopy
  LONG IF WINDOW (_efClass) < 0
    pictH& = FN GetPICTHandle&
    LONG IF pictH& <> _nil
      scrapErr% = FN DataHandleToScrap (pictH&, _"PICT", _true)
      DEF DISPOSEH (pictH&)
    END IF
  END IF
END FN
```

DataHandleToScrap starts by accepting three parameters, a handle containing the data to place on the clipboard, in this case PICT, and a flag that allows us to clear the clipboard or just add to the desk scrap.

It makes sure it has a valid handle, then examines the zeroClipBoard% flag to determine whether to clear the clipboard using FN ZeroScrap. Next, it gets the handle size, locks it from moving in memory with FN Hlock, and calls PutScrap to copy the data into the clipboard. It finishes by unlocking the handle with FN HunLock and returns any error it encountered.

PROGRAM 140. Sending data to scrap.

```
CLEAR LOCAL
LOCAL FN DataHandleToScrap (dataH&, dataType&, zeroClipBoard%)
  LONG IF dataH& <> _nil
    LONG IF zeroClipBoard% <> _nil
      scrapH& = FN ZEROSCRAP
    END IF
    sizeOfH& = FN GETHANDLESIZE (dataH&)
    osErr% = FN HLOCK (dataH&)
    LONG IF osErr% = _noErr
      osErr% = FN PUTSCRAP (sizeOfH&, dataType&, [dataH&])
      osErr% = FN HUNLOCK (dataH&)
    END IF
  END IF
END FN = osErr%
```

Cut Picture

When the user chooses **Cut** from the **Edit** menu, `EditCut` is called. It checks to make sure we are in the picture field, then calls `EditCopy` to copy the picture data from the picture field to the desk scrap. Then, since it's cutting the picture from the record, it clears the picture field with `EDIT$`, sets the `dbPictID%` field of the employee record to nil, and disposes of the global picture handle `gPictH&`. This subroutine is shown in Program 141.

PROGRAM 141. EditCut subroutine.

```
CLEAR LOCAL
LOCAL FN EditCut
  LONG IF WINDOW (_efClass) < 0
    FN EditCopy
    EDIT$ (_dbPhotoFLD) = ""
    gEmployee.dbPictID% = _nil
    DEF DISPOSEH (gPictH&)
  END IF
END FN
```

Clear Picture

`EditClear`, shown in Program 142, is exactly like `EditCut` with the exception that it doesn't copy the picture field's contents to the desk scrap.

PROGRAM 142. EditClear subroutine.

```
CLEAR LOCAL
LOCAL FN EditClear
  LONG IF WINDOW (_efClass) < 0
    EDIT$ (_dbPhotoFLD) = ""
    gEmployee.dbPictID% = _nil
    DEF DISPOSEH (gPictH&)
  END IF
END FN
```

Copy Record

One final export option we should look at involves copying an entire record to the Clipboard. This enables the user to place all of the data in a single record into another program in one step. The most common format has each field in our record separated by a tab so that the exported information appears in a database, spreadsheet or word processing document in the proper alignment.

`EditExportRecord`, shown in Program 143, starts by calculating the total length of all entries in the current record. As soon as it has that, it creates a handle to hold all of the data. Once it has a valid handle, it locks it in memory with `FN HLock`, then cycles through each record field, extracting the field text, appends a tab onto the end and inserts it into the handle. A variable called `offset&` keeps track of the next position to place data. When the final field is read (fax number in *SimpleBase*), it appends a carriage return instead of a tab, then calls `DataHandleToScrap` with the type `TEXT` to place the data onto the desk scrap.

Importing Data

Exporting picture data isn't the whole game. We also need to import data via the clipboard in order to be totally conversant with other programs. The routine that does this for us is called `FN ScrapToDataHandle&` and can be seen in Program 144.

`FN ScrapToDataHandle&` starts by accepting one parameter, the type of scrap to search for in the desk scrap. In our case, that will be `PICT`. It starts by creating a new handle to hold the data, then uses the Toolbox function `GetScrap` to see if any `PICT` data is on the desk scrap. If there is, it's copied to the empty handle (which isn't empty anymore) and returns it to the calling routine.

PROGRAM 143. Copy record to scrap.

```
CLEAR LOCAL
LOCAL FN EditExportRecord
  FOR count% = _dbNameFLD TO _dbFaxFLD
    calcHSize% = calcHSize% + LEN(EDIT$(count%)) + 1
  NEXT count%
  offset% = 0
  recordH& = FN NEWHANDLE (calcHSize% + offset%)
  LONG IF (recordH& <> 0) AND (SYSERROR = _noErr)
    osErr% = FN HLOCK (recordH&)
    LONG IF osErr% = _noErr
      FOR count% = _dbNameFLD TO _dbFaxFLD
        tmp$ = EDIT$(count%)
        LONG IF count% < _dbFaxFLD
          char$ = CHR$(_tab)
        XELSE
          char$ = CHR$(_cr)
        END IF
        tmp$ = tmp$ + char$
        size% = LEN (tmp$)
        BLOCKMOVE @tmp$+1, [recordH&] + offset%, size%
        offset% = offset% + size%
      NEXT count%
    END IF
    osErr% = FN HUNLOCK (recordH&)
    osErr% = FN DataHandleToScrap (recordH&, _"TEXT", _true)
    DEF DISPOSEH (recordH&)
  END IF
END FN
```

PROGRAM 144. Importing data from the scrap.

```
CLEAR LOCAL
LOCAL FN ScrapToDataHandle& (scrapType&)
  scrapH& = FN NEWHANDLE (0)
  LONG IF scrapH& <> _nil
    scrapSize& = FN GETSCRAP (scrapH&, scrapType&, offset&)
    LONG IF scrapSize& <= 0
      DEF DISPOSEH (scrapH&)
    END IF
  END IF
END FN = scrapH&
```

Paste Picture

When **Paste** is called and the picture field is active, the `EditPaste` subroutine in Program 145 makes sure it's in the picture field, then calls `FN ScrapToDataHandle&` to return a handle containing picture data to us. If the handle comes back valid, it disposes of the current global picture data in `gPictH&` using `DEF DISPOSEH`, saves the new picture handle to the global handle, and places it into the picture field using `EDIT$`. Finally, it assigns the `gOpenRecord%` to `gEmployee.dbPictID%` so that it will be saved to the employee file.

PROGRAM 145. EditPaste subroutine.

```
CLEAR LOCAL
LOCAL FN EditPaste
  LONG IF WINDOW (_efClass) < 0
    pictH& = FN ScrapToDataHandle& (_"PICT")
    LONG IF pictH& <> _nil
      DEF DISPOSEH (gPictH&)
      gPictH& = pictH&
      EDIT$ (_dbPhotoFLD) = &gPictH&
      gEmployee.dbPictID% = gOpenRecord%
    END IF
  END IF
END FN
```

Paste Records

Of course, if we can export an entire record, we should also be able to import a record that is formatted properly. We know we are importing a record, and all the program has to do is parse the text handle it finds on the clipboard into separate fields, placing them appropriately. The entire `EditImportRecord` function is shown in Program 146.

It starts by calling `ScrapToDataHandle&` to see if there is any `TEXT` on the desk scrap. If there is, it sets up some string pointers (for greater speed) and then assigns a Tab to a string called `char$`. This is what separates each field in the record and what we'll search for to determine where to parse the field data.

Next, it uses a Toolbox function called `Munger` to search the text handle. `Munger` is great for searching, inserting, replacing, or deleting text from handles. Think of it as `MID$`'s big brother. `Munger` wants as parameters: a handle containing text, a position to begin the search, a pointer to a search string, the search string's length, a pointer to a replacement string and its

length. Since we are only searching, we can leave the replacement string and length nil. And by manipulating where the search starts, we can walk through the handle looking for each field separately.

Note that the pointers to the search and replacement strings must contain the address to the first character of each string, not their length bytes. That's why a 1 is added to each pointer, to skip the length byte.

If `Munger` returns a negative value, that means it didn't find anything to match the search criteria (our Tab), so we inform the user with another alert that the text they are trying to paste is not a record and exit the function. If

PROGRAM 146. Copy record from scrap.

```
CLEAR LOCAL
DIM   tmp$
DIM 3 char$
LOCAL FN EditImportRecord
  scrapH& = FN ScrapToDataHandle& (_"TEXT")
  LONG IF scrapH& <> _nil
    strPtr& = @tmp$
    charPtr& = @char$ + 1
    char$ = CHR$(_tab)
    startPos& = _nil
    offset& = FN MUNGER (scrapH&, startPos&, charPtr&, 1, _nil, _nil)
    LONG IF offset& > _nil
      fieldID% = _dbNameFLD
      DO
        size% = offset& - startPos&
        POKE strPtr&, size%
        BLOCKMOVE [scrapH&] + startPos&, strPtr&+1, size%
        EDIT$ (fieldID%) = tmp$
        INC (fieldID%)
        startPos& = offset& + 1
        offset& = FN MUNGER (scrapH&, startPos&, charPtr&, 1, _nil, _nil)
      UNTIL (offset& < 0) OR (fieldID% = _dbFaxFld)
      size% = FN GETHANDLESIZE (scrapH&) - startPos&
      POKE strPtr&, size%
      BLOCKMOVE [scrapH&] + startPos&, strPtr&+1, size%
      EDIT$ (fieldID%) = tmp$
    XELSE
      item% = FN NOTEALERT (_noRecordALRT, 0)
    END IF
    DEF DISPOSEH (scrapH&)
  END IF
END FN
```

Munger finds a match it returns the offset into the handle one byte after the match position.

Once we have our first match, the real work begins. In a DO/UNTIL loop we calculate the size of the field's data by taking the offset& - startPos&, then we POKE that into a temporary string variable defined earlier, followed by a BLOCKMOVE of the text from the startPos& to offset& position into the string variable. Next, the temporary string is placed into the waiting edit field, the fieldID% is incremented, startPos% is updated to one byte past offset&, and we call Munger again using the new offset position.

This cycle continues until Munger fails (offset& < 0). At this time we know only the final field's data remains, so we calculate the size of the handle minus the startPos&, and repeat the POKE and BLOCKMOVE of the final field's data into the temporary string and into the edit field. The last act is to dispose of the handle using DEF DISPOSEH.

Cooldown

And that's all there is to cutting, copying, clearing, and pasting picture data in *SimpleBase*. Now that wasn't so bad, was it?

In this chapter we learned about the desk and TE scraps (more commonly referred to as the Clipboard) and their major differences. We also saw how to manipulate the Edit menus so that the runtime has control when manipulating text data, and we have control for pictures and record data. We learned how to interpret the picture data in a picture field, and how to copy, cut, and paste it into other picture fields as well as other programs. Finally we saw how easy it is to manipulate records of data to enable copying of complete records from place to place in the employee database.

Use of these techniques will make your programs more user friendly and enable you to provide the editing support your users desire.

Printing

Warm-up

We're nearing the end, this chapter explains how to print text and graphics from any program. In it you will learn:

- How to print to any attached printer,
- How to handle the standard print loop, and
- How to handle printer errors.

Printing on the Macintosh is remarkably easy to do if you keep one thing in mind: sending text and graphics to a printer is exactly like printing to a program window.

The Print Manager

The Print Manager is the Toolbox manager that handles printing requests on the Macintosh. It's the Print Manager's responsibility to provide a common interface to the print driver for the attached printer. A **print driver** is software that translates your program's printable output into a language the printer can understand. Print drivers are selected by the user with the System's Chooser and include files like LaserWriter and ImageWriter.

Normally, a program should never worry about which printer is connected, you call the same print routines. This makes it very easy to write print routines in a program that work on any Chooser compatible printer. Some of these routines are described below.

The Print Record

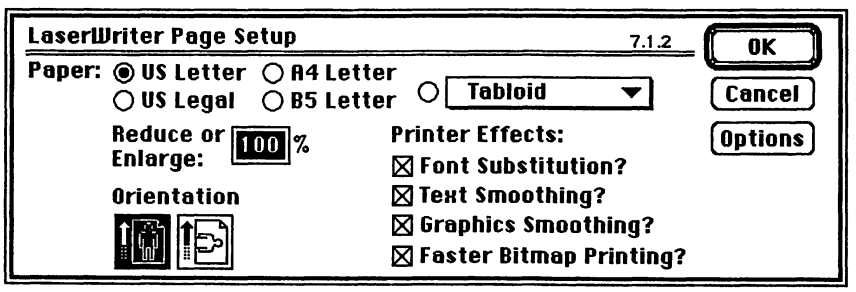
All of the formatting information for a printer is stored in a print record. This information includes things like the paper size, the paper's printable area, the resolution, and a host of user options including the number of copies to print, paper size, and many more. Options are set using two printer dialogs, one for style information, and one for print information.

Page Setup

The Page Setup dialog enables users to specify page dimensions, page orientation, and for laser printers, special effects like graphics text smoothing.

You access the Page Setup dialog in a program using the DEF PAGE statement. Users make their selections and click the **OK** button to save the changes to the print record. Use the PRCANCEL function to determine if the **Cancel** button was selected.

FIGURE 91. Printer style dialog.



Print...

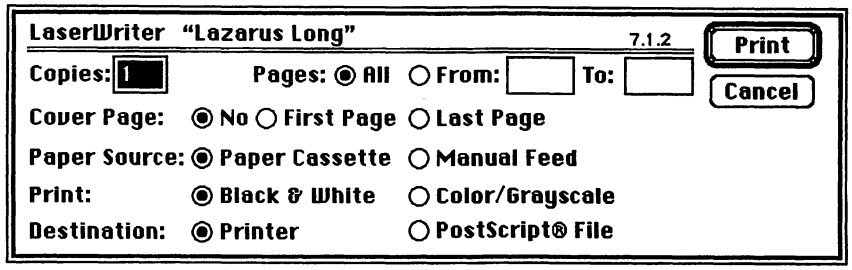
The Print dialog enables the user to control the number of copies, page selections, type of printing, and several other immediate print job requirements. Unless cancelled, the normal sequence is to print the document after viewing this dialog. It's at this point that your program takes charge and handles the actual work of printing.

To display the print job dialog, use DEF LPRINT. Again, use PRCANCEL to determine if the user clicked the **Cancel** button.

Print Record

The current print record is accessed using FB's PRHANDLE function. The information stored therein should not be considered valid until both the Page

FIGURE 92. Printer job dialog.



Setup and Print dialogs have been okayed by the user. We'll soon see how to extract relevant portions of that information as required by the program.

- *See Inside Macintosh, Vol. 2, or the FB Reference manual for a description of the Print record.*

Routing Output

Once it's determined that the user wants to print, i.e. PRCANCEL is false, use the ROUTE statement to direct all subsequent text and graphic commands to the Print Manager. Normally, all text and graphic commands go to the current output window, but to send them to the printer, use ROUTE _toPrinter. This output includes EDIT FIELD, PICTURE FIELD, PICTURE, BUTTON, and SCROLL BUTTON statements too.

As soon as the print job is complete, use ROUTE _toScreen to return output to the current window. Additionally, use of ROUTE in the middle of a print job enables you to send output to the printer and also display status information in a window.

- *Note that you can use ROUTE for the modem ports too. See the FB Reference manual for details.*

Page Information

To all intents and purposes, once you've routed output to a printer, you can treat the paper as just another output window. This flexibility allows you to write a single routine that will print a report to the screen, and output the same report to any printer just by changing the ROUTE statement.

Since you set the size of the page in the Page Setup dialog by choosing the paper type, your program should be able to detect the change and make adjustments to provide the best printout possible. For example, you can

PROGRAM 147. Getting the page size.

```
ROUTE _toPrinter
printHt% = WINDOW (_height)
printWd% = WINDOW (_width)
ROUTE _toScreen
```

determine the actual printable area of the page using the WINDOW functions shown in Program 147.

Additionally, use the PRHANDLE function which returns a handle to the actual print record that describes the current print job. There is a host of information stored inside it, but most programs won't require that amount of control. But, some programs do require more information, so the example in Program 147 shows how to use PRHANDLE to get the starting and ending page numbers, as well as the number of copies.

PROGRAM 148. Other printer information.

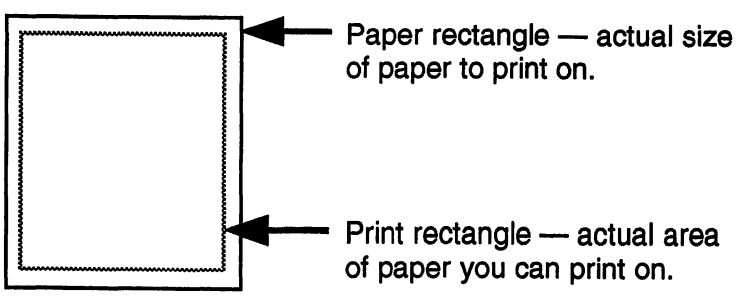
```
prHndl& = PRHANDLE
numCopies% = prHndl&..iCopies%
startPg% = prHndl&..iFstPage%
lastPg% = prHndl&..iLstPage%
vertRes% = prHndl&..iVRes%
horzRes% = prHndl&..iHRes%
```

Additionally, while treating the printer as a window you can output text using different fonts, text sizes and styles, as well as colors and output graphics using all of the FB and Toolbox commands.

Printing

FutureBASIC offers a lot of control over printing text. What follows is the three most common commands used to output text to a printer.

FIGURE 93. Print rectangle on paper.



LPRINT

LPRINT is the old-fashioned means of outputting text to a printer. When using LPRINT, the Print Manager is bypassed, and the character data is sent directly out the chosen serial port to the printer. Formatting your output is difficult since you must send special printer code sequences to the printer to format the subsequent text. LPRINT uses the printer's native fonts which has the benefit of higher speed.

LPRINT does have some other problems. It may not be reliable over networks, doesn't like some laser printers, and since different dot-matrix printers offer different features, the code sequences to format output may differ from printer to printer. Also, LPRINT is slow. It actually creates and closes a new printer port each time it is called. All-in-all, these non-features of LPRINT make it a poor choice for outputting anything to a printer.

PRINT%

The standard PRINT% statement is the preferred method of outputting text to a printer (PRINT and PRINT@ are close behind). For all intents and purposes PRINT% functions exactly as when outputting strings to a window at a precise pixel location. This makes it very easy to write print routines that serve the dual purpose of printing exactly the same to both a window and a printer. A small example of this is shown in Program 149. Additionally, it's probably a bit faster than using the edit field method described next.

PROGRAM 149. Dual print routine.

```
LOCAL FN DualPrint (whereTo)
  LONG IF whereTo = _toPrinter
    ROUTE _toPrinter
  END IF
  ' ••• PRINT YOUR FORMATTED DATA HERE
  ROUTE _toScreen
END FN
```

Once output is routed to the printer, you can use any of the standard formatting statements to change the font, size, style, and mode of the outputted data. Again, just like you can in a window.

EDIT & PICTURE FIELDs

When output is routed to a printer, it can be treated exactly as if it were a window. This means that you can create both edit and picture fields[1] in this

1. And buttons and scroll buttons as well.

printer output window. These edit and picture fields can have all the attributes of their normal window brothers, including frames, styled text, alignment, etc. For outputting large quantities of styled text or pictures this technique works very well. In fact, we'll use a PICTURE FIELD to print the employee's picture along with their other personal data.

This method may be a bit slower than PRINT%, but the benefits gained far outweigh the disadvantages.

Graphics

Finally, when it comes to outputting graphics to a printer, you can use all of FB's graphic commands: PLOT, BOX, PLOT TO, BOX FILL, CIRCLE, CIRCLE FILL, PEN, and COLOR, as well as all Toolbox graphic commands.

Regular Exercise

Now that we understand more about printing, it's time to begin adding a printing capability to *SimpleBase*. You'll want to insert these print routines near the top of the main file. See the Appendix for the complete *SimpleBase* listing.

To do that we'll create another include file called *Printing.Incl*. In it we will place all of the print routines. Just as before, here are the directions to get started.

Set *Printing.Incl* up as an include file with the standard program headers used in the other include files. At the very bottom of the new file add:

```
gDoPrintPtr& = @FN pDoPrinting
```

Then in *SimpleBase.glbl* add:

```
DIM gDoPrintPtr&
```

And in *Project.Incl* add:

```
DEF FN DoPrinting (readRecPtr&) USING gDoPrintPtr&
```

Finally, in the *SimpleBase.Incl* file, delete the old DoPrinting subroutine. Remember to save all of your files. The starting source for the *Printing.Incl* is shown in Program 150.

A Standard Print Loop

We'll begin by implementing a standard printing loop. This is the core printing function that you can drop into a program and modify. It handles most of the requirements for setting up and disposing of a print job, all you do is add

PROGRAM 150. Printing.Incl file.

```
' --- HEADER -------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels
' --- CONSTANTS ----------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS
' --- GLOBAL FUNCTIONS ---------------------------
LOCAL FN DBReadRecordTemplate
END FN
' --- FUNCTIONS ----------------------------------
LOCAL FN PrintRecord
END FN
LOCAL FN PrintManyRecords
END FN
LOCAL FN pDoPrinting (readRecPtr&)
END FN

' --- GLOBAL POINTER -----------------------------
gDoPrintPtr& = @FN pDoPrinting
```

your program's specific printing code. We'll explain it as we build it so that if you need to modify it later, you'll have a place to start.

The print loop starts from when the user has selected **Print** from the **File** menu. Naturally, the user expects a print job dialog, so use DEF PAGE to display one. Next, it checks to see if the user selected the **Cancel** button using PRCANCEL.

PROGRAM 151. Standard print routine.

```
LOCAL FN pDoPrinting
  DEF LPRINT
  LONG IF PRCANCEL = 0
    CURSOR _watchCursor
    ROUTE _toPrinter
    ' ••• place custom print routine here
    ROUTE _toScreen
    CLOSE LPRINT
    CURSOR _arrowCursor
  END IF
END FN
```

If the user cancels the dialog, control returns to the program's Main Loop. However, when **Print** is chosen, the routine sets the cursor to a watch and uses ROUTE to send all text and graphics to the printer. We'll skip the custom print routines for now since they can be anything. When control returns from the program's custom print routine, output routes back to the screen, and the Print Manager is closed with CLOSE LPRINT.

The Custom Print Routine

Now that we have our standard print loop, let's jazz it up a bit and make it useful for *SimpleBase*.

Our custom print routine in Program 152 begins with a DEF LPRINT statement that enables the user to choose the number of copies to print. Once **Print** is chosen, we store the current record number for later use. Open the database file's resource fork with USR OPENRFPERM to allow us access to the employee pictures stored there. A change of cursor to show we're busy and the ROUTE statement will send all subsequent output to the printer.

Since we have three choices in the Print window (current, all, or selected records), we use a SELECT structure to handle the user's choice. Depending on the value of gPrintFlag%, we implement one of two subroutines: PrintRecord, or PrintManyRecords.

To print the current record only, we pass that job off to PrintRecord which has the responsibility of printing a single employee record. Upon completion, we use CLEAR LPRINT to tell the Print Manager we're done with that page and to print it. Exiting the SELECT structure routes output back to the screen with ROUTE, then closes the Print Manager with CLOSE LPRINT. If resRef% is valid it closes the database file with CloseResFile. Finally, we reset the value of gOpenRecord%, read the saved record (in case it changed as we'll see later), and show the arrow cursor to let the user know we're done.

To print multiple records, including both all and selected records, everything up to the SELECT structure remains the same except the value of gPrintFlag%. To print all records, or a selected few, the routine calls PrintManyRecords and passes it two parameters, the first and the last record number to print. "Ah ha", you say, finally you can see why it was important to save the current value of gOpenRecord%. Since PrintManyRecords uses gOpenRecord% to access the specified records, we must have some way of returning the user to the record they started at. In the case of printing all records, we pass parameters of zero for first record and gMaxRecInFile% for the final record. To print only a chosen few, we pass gPrFirstRec% and gPrLastRec% instead. The same routine handles both variations.

PROGRAM 152. Custom printing records routine.

```
LOCAL FN pDoPrinting (readRecPtr&)
  DEF LPRINT
  LONG IF PRCANCEL = 0
    oldRecNum% = gOpenRecord%
    resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, _fsCurPerm)
    CURSOR _watchCursor
    ROUTE _toPrinter
    SELECT gPrintFlag%
      CASE _thisRecBTN
        FN PrintRecord (10)
        ROUTE _toScreen
        CLEAR LPRINT
      CASE _allRecBTN
        FN PrintManyRecords (1, gMaxRecInFile%, readRecPtr&)
      CASE _selectRecBTN
        FN PrintManyRecords (gPrFirstRec%, gPrLastRec%, readRecPtr&)
    END SELECT
    ROUTE _toScreen
    CLOSE LPRINT
    IF resRef% THEN CALL CLOSERESFILE (resRef%)
    gOpenRecord% = oldRecNum%
    FN DBReadRecordTemplate USING readRecPtr&;
    CURSOR _arrowCursor
  END IF
END FN
```

That's the controller subroutine that handles all of our required printing tasks. Now let's look at the individual routines that print the employee records.

Global Template

You may have noticed that in the subroutine `pDoPrinting` we pass it a pointer as a parameter and later on use it in the `FN DBReadRecordTemplate` call. What's going on here?

What's going on is that the `DBReadRecord` subroutine is located in the *SimpleBase.Incl* file which can't be seen from *Printing.Incl* directly. We could have placed it into the *Project.Incl* but thought we'd demonstrate another means of calling a subroutine with `FN USING`. In this version we pass the address to the subroutine we want to access as a parameter. In the include file that is called, we create a template function, just as we do in the *Projects.Incl*.

Since DBReadRecord is stored in *SimpleBase.Incl*, it's an easy matter to use @FN to get its address and pass it onto pDoPrinting. pDoPrinting gets the address and uses it to call DBReadRecord from itself via DBReadRecord-Template when required. A nice symbiotic relationship.

So in *SimpleBase.Incl*, the call is made like this:

```
FN DoPrinting (@FN DBReadRecord)
```

Printing a Single Record

Now let's examine the actual printing routine. I'm sure you thought we'd never get there. This is the real workhorse of the print loop. It's here that we take the actual data, format it correctly, and then print it.

We start by determining what the layout of each record should look like when printed. Since *SimpleBase* only has a few fields, it was easy to design a layout that had an employee's picture on the left, the field titles in the middle, and the actual data next to the field titles. This layout has two benefits: one it looks good, and two, it allows four employee records to appear on a single printed page which can save a lot of paper.

The subroutine PrintRecord accepts one parameter called pgVOffset%. This parameter is used to specify which of four record positions on the page the current record will be printed. Another important value used here, _gutterAdj, is defined as a constant. This constant defines how much gutter space a page should have. Adjust it to suit your preferences, or change it to a global value that the user can modify themselves. Of course, you'll need to add a preferences window to accomplish this.

The subroutine PrintRecord starts by defining a couple of coordinate offsets to correctly place the various elements of an employee record on the page. It loops through the field titles stored in the STR# _printerSTR and prints them to the page while adjusting the vOffset% variable. Next, a series of PRINT% statements print the data in the gEmployee fields using xOffSet% and _gutterAdj as well as pgVOffset%.

We set the position of the employee picture, draw it using a PICTURE FIELD, and finish by adding a gray dividing line after the employee record.

PROGRAM 153. Print a page routine.

```
LOCAL
DIM rect.8
LOCAL FN PrintRecord (pgVOffset%)
  xOffSet% = 150
  vOffset% = 15
  TEXT _geneva, 9, 1
  ' ••• PRINT FIELD TITLES
  FOR count% = _dbNameFLD TO _dbFaxFLD
    tmp$ = UCASE$ (STR# (_dbEntryWIND_fieldSTR, count%))
    PRINT%(xOffSet% + _gutterAdj, pgVOffset% + vOffset%) tmp$
    vOffset% = vOffset% + 15
  NEXT count%
  ' ••• PRINT FIELD DATA
  TEXT "_geneva, 12", 0
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 15) gEmployee.dbName$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 30) gEmployee.dbAddr$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 45) gEmployee.dbCity$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 60) gEmployee.dbMyState$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 75) gEmployee.dbZip$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 90) gEmployee.dbPhone$
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% +105) gEmployee.dbFax$
  ' ••• PRINT PICTURE & SEPERATOR
  CALL SETRECT (rect, _gutterAdj,pgVOffset%, 130 + _gutterAdj, pgVOffset%+152)
  PICTURE FIELD #100, %gEmployee.dbPictID%, @rect, _statFramed, _cropPict
  PEN ,,,,3
  PLOT 0, pgVOffset% + 165 TO 600, pgVOffset% + 165
  PEN ,,,,0
END FN
```

Printing Multiple Records

Once we have the record printing routine finished and tested, it's not difficult to add another routine shown in Program 154 to print multiple records. The routine only requires two parameters: the number of the first and last records to print. In this case, the first record to print is always record number one, and the final record is defined by `gMaxRecInFile%`.

After some initial setup, the routine cycles through getting each record into memory using `DBReadRecord`, then printing it using the custom printing subroutine previously described. After it returns from printing one record, it increments its control variables and repeats until all specified records have printed. Once the record count exceeds the last record specified, control drops out of the loop and eventually returns to the Main Loop.

In the loop, we have some special code that determine how many records have already been printed. When four records have printed to the page, a page number is then printed at the page bottom and the page is closed. This causes the Print Manager to close the page in memory and begin processing it for printing. Meanwhile the program can begin printing the next page of records.

PROGRAM 154. Print multiple records routine.

```
LOCAL FN PrintManyRecords (firstRec%, lastRec%, readRecPtr&)
  pgVOffset% = 10
  pageNum% = 1
  recCount% = 0
  DO
    gOpenRecord% = firstRec%
    FN DBReadRecordTemplate USING readRecPtr&;
    FN PrintRecord (pgVOffset%)
    INC (firstRec%)
    INC (recCount%)
    LONG IF (recCount% MOD 4) = 0
      PRINT%(_gutterAdj, pgVOffset% + 180) "PAGE#";pageNum%
      INC (pageNum%)
      pgVOffset% = 10
      IF recCount% < lastRec% THEN CLEAR LPRINT
    XELSE
      pgVOffset% = pgVOffset% + 180
    END IF
  UNTIL firstRec% > lastRec%
END FN
```

Printing Selected Records

Finally, the last printing option to include is one that prints from a start record to a final record. The previously written `PrintRecord` and `PrintManyRecords` routines will again be used to provide this handy feature. All we need to do is pass the `gPrFirstRec%` and `gPrLastRec%` values to `PrintManyRecords` functions, and voilá, we have a custom printing routine that prints any record or number of records we choose.

The key is gathering the users input data from the edit fields in the Print window. Of course, that task falls to `WindowCapture` as shown in Program 155. We add another small but handy routine called `CheckRange%` that ensures our starting and ending values fall within accepted record number boundaries

PROGRAM 155. Print selected records.

```
LOCAL FN CheckRange% (current%, minRange%, maxRange%)
  IF current% < minRange% THEN current% = minRange%
  IF current% > maxRange% THEN current% = maxRange%
END FN = current%

LOCAL FN WindowCapture
  SELECT wndID%
    CASE _printWIND
      gPrLastRec% = VAL(EDIT$(_lastPrFLD))
      FN CheckRange% (gPrLastRec%, 1, gMaxRecInFile%)
      gPrFirstRec% = VAL(EDIT$(_firstPrFLD))
      FN CheckRange% (gPrFirstRec%, 1, gPrLastRec%)
  END SELECT
END FN
```

Closing the Print Manager

Just as there is a sequence to opening the Print manager to deal with your printing task, there's also a recommended method of finishing it.

The big finish occurs when there are no more pages to print. At that point we use `CLOSE LPRINT` to close the Print manager. This tells the Print manager that there are no more pages to print, enabling it to close down.

Right after that, always ensure that you `ROUTE` output back to the screen, and that you close all of your own private data structures, print message windows, etc.

Peak Performance

There may come a time when you want to do something fancier with the printer. The following are some routines that may be useful to you.

Get Printer Name

Occasionally your program may need to know which printer is currently selected. You may need to display the printer type, or simply use the name in a dialog or alert. In either case, the following routine will locate the name of the currently chosen printer as selected by the Chooser desk accessory.

- *In reality, a program should not concern itself with which printer is currently selected since a print routine should work with any attached printer.*

The following routine searches for the name of the currently chosen printer on the computer. That information is stored in the System file as a STR resource with an ID -8192. The routine searches all open resource forks for the matching string. Since negative STR values are reserved by Apple for system resources, this string will only be found in the System file. The routine is shown in Program 156.

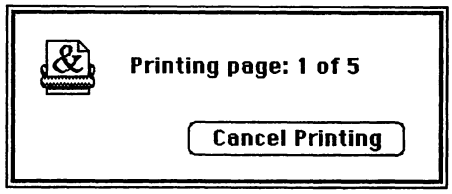
PROGRAM 156. Get Printer Name.

```
CLEAR LOCAL MODE
DIM 63 printerName$
LOCAL FN GetPrinterName$
  resH& = FN GETSTRING (-8192)
  LONG IF resH& = 0
    printerName$ = "unknown"
  XELSE
    printerName$ = PSTR$([resH&])
    CALL RELEASERESOURCE (resH&)
  END IF
END FN = printerName$
```

Getting a Printer Icon

Yet another piece of information you might require is the actual icon representing the currently chosen printer. This is great for the users since they can see immediately which printer they have chosen (if they have several) and also makes them wonder how the program knew which printer to display.

FIGURE 94. A standard print cancel dialog.



In Program 157, we make use of the `GetPrinterName` function to locate the name of the printer extension file. We also determine which operating system the program is operating under and search in the appropriate folder for the printer extension. We do some fancy folder dancing to get the correct `wdRefNum%` and then open the printer file's resource fork. Once open, we extract the printer icon, close up shop, and return the icon's handle.

- *Under System 6, all printer files reside in the System Folder itself, under System 7 printer files are located in the Extensions folder within the System Folder.*

Well, the first order of business is to get the chosen printer's name as demonstrated in "Get Printer Name". Next, we use the `SYSTEM` function to return the working directory reference number of the System Folder in the `wdRefNum%` variable.

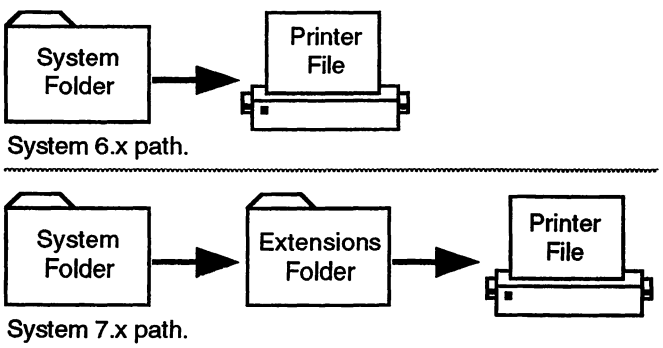
Now we do some more fancy folder work. If the program is executing under System 6 there is nothing more to do because printer files live in the System Folder, so the next routine is skipped. However, if it's operating under System 7 we must switch to the Extensions folder to locate our printer file. We do that using two variations of the `FOLDER` function. The first uses the System's volume reference number to open the System Folder. The second `FOLDER` call searches the System Folder to see if a folder called Extensions lives there. If it does, it opens that folder and returns its volume reference number in `vRefNum%`.

PROGRAM 157. Get Printer Icon.

```
CLEAR LOCAL MODE
LOCAL FN GetPrinterIcon
  printerName$ = FN GetPrinterName$
  oldResRef% = FN CURRESFILE
  wdRefNum% = SYSTEM (_sysVol)
  LONG IF SYSTEM (_sysVers) > 699
    wdRefNum% = FOLDER ("", wdRefNum%)
    wdRefNum% = FOLDER ("Extensions", 0)
  END IF
  resRef% = USR OPENRFPERM (printerName$, WDRefNum%, _fsCurPerm)
  LONG IF resRef = 0
    BEEP : BEEP
  XELSE
    printerIcnH& = FN GET1INDRESOURCE (_"ICN#", 1)
    LONG IF printerIcnH& = 0
      BEEP : BEEP
    XELSE
      CALL DETACHRESOURCE (printerIcnH&)
    END IF
    IF resRef% THEN CALL CLOSERESFILE (resRef%)
  END IF
  CALL USERESFILE (oldResRef%)
END FN = printerIcnH&
```

The search path we use is illustrated in Figure 95. The top portion of the diagram shows that printer files for System 6 are stored in the System Folder. On the bottom, printer extensions reside in the Extensions folder within the System Folder. In either case, the routine now attempts to open the printer file called `printerName$` in the folder specified by `wdRefNum%` using `USR OPENRFPERM`. A valid `resRef%` number (non-zero) indicates success.

FIGURE 95. Printer file paths.



With a valid `resRef%` we search the printer file for the first `ICN#` resource stored in its resource list using the Toolbox function `Get1IndResource`. `Get1IndResource` grabs the first resource of the requested type in the printer file and returns a handle to the icon data. Once it has the handle, it detaches it from the file allowing us to close the printer file with `CloseResFile`. The icon handle is then returned to the calling routine.

We can now display it in a window, a dialog, or an alert with `PlotIcon` and leave the user wondering how we figured it out. The small example in Program 158 shows how to do this.

PROGRAM 158. Display printer icon.

```
DIM rect.8
CALL SETRECT (rect, 20, 20, 52, 52)
WINDOW 1 : CLS
printerIcnH& = FN GetPrinterIcon
LONG IF printerIcnH&
  CALL PLOTICON (rect, printerIcnH&)
  DEF DISPOSEH (printerIcnH&)
END IF
STOP
```

Cooldown

In this chapter we learned about the various methods of sending data to a printer. We saw how some useful printing statements may not provide the amount of control or speed you require from your program. We also learned how to create simple yet effective routines to handle a multitude of printing tasks, from individual records, to all records, and selected records.

With the information provided in this chapter you are well on your way to becoming a printing master. In the next chapter, we'll create many of the resources necessary to make *SimpleBase* a unique application in the eyes of the *Finder* and everyone else.

CHAPTER 18

Application Resources

Warm-up

We're nearing the end of creating *SimpleBase*. What follows are some addition resources required by applications on the Macintosh. Here you'll learn all about:

- The `BNDL` resource,
- `FREF` resources,
- `ICN#` resources,
- The `signature` resource,
- The `vers` resource,
- The `SIZE` resource, and
- Special System 7 resources.

There are several resource types required by any program for it to be recognized as a unique application by the *Finder*. What follows are explanations of these various resources and how they relate to an application and to each other.

- *Note that the following resources are required under both System 6 and System 7 for all applications.*

The BNDL Resource

The BNDL (bundle) resource identifies all *Finder*-related resources associated with the application program. These resources include the ICN#, FREF, and signature resources.

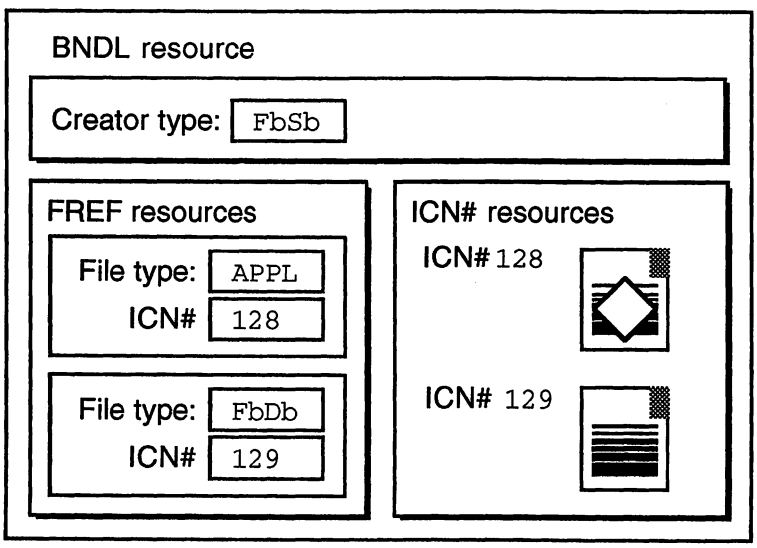
The BNDL resource links these three resources together into, well, a bundle. To begin, the BNDL resource identifies the application's signature resource (4-character type) to the *Finder*. This enables the *Finder* to link documents with specific file types to the application that created them. I.e., the user can double-click on a document and the *Finder* will search out and open the application that created the document.

Next the BNDL links a FREF resource to an ICN# resource. The FREF (file reference) contains a 4-character file type, and an index number to the ICN# resources. Typically, the first FREF relates to the application itself, and the first ICN# in the index is the application icon. The next FREF represents a single file type the application can save to disk and its associated ICN# index number. Another FREF resource is required for each file type associated with the application. You can see this resource relationship in Figure 111.

The Signature Resource

A signature resource is an application unique resource type that enables the *Finder* to identify and launch the correct application when the user double-clicks on a document created by that application. The signature resource is a

FIGURE 96. BNDL resource relationships.

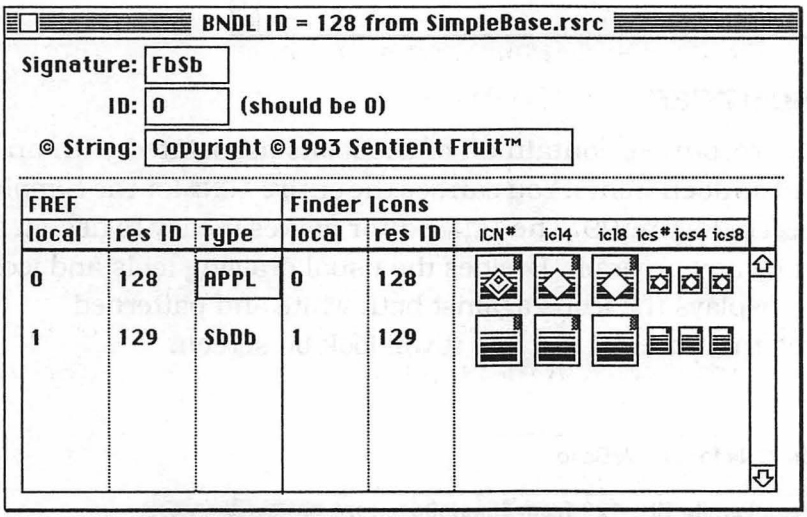


4-character type commonly referred to as the creator type. For example, *FutureBASIC* has a creator type of ZBAS and *ResEdit* has one of RSED. You can designate the signature type in the BNDL editor as shown in Figure 97.

Your programs should have their own unique creator types. If a program has the same creator type as another, the *Finder* can become confused and display the wrong icons on both the application and its documents.

Programs specify a file type for a file using the DEF OPEN statement as previously shown in the chapter "Files". You can determine a document file type using FILES$ or FINDERINFO as demonstrated in the chapter "Final Touches".

FIGURE 97. BNDL editor in ResEdit.



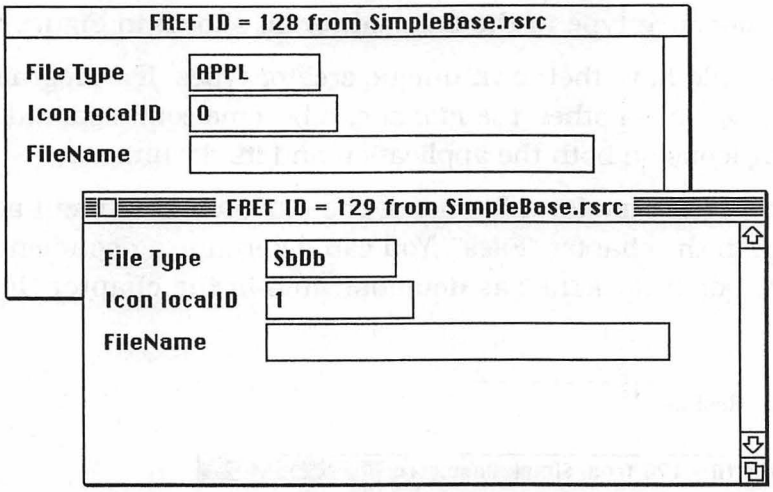
- *Note, to ensure that your applications signature doesn't conflict with another application, register your application's signature with Apple Computer, Inc., at Macintosh Developer Technical Support.*

FREF Resources

FREF (file reference) resources are used to link specific file types with the icon used by the *Finder* to display it on the screen. For each file type an application can create, it should have a FREF resource of that type.

A FREF resource contains three items: a 4-character file type, a local ICN# ID, and an empty string (never implemented by Apple). File types can be a common one like TEXT or PICT, or be unique to your application. The two FREFs used by *SimpleBase* are shown in Figure 98.

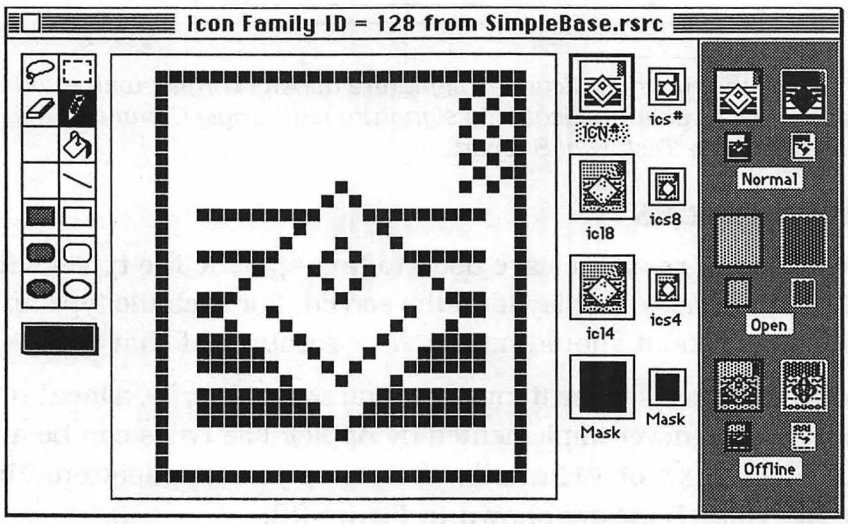
FIGURE 98. The FREF resources used by SimpleBase.



ICN# Resources

ICN# (icon list) resources contain all of the icons associated with an application and its documents. You can see the entire ICN# for the *SimpleBase* application icon in Figure 99. The ICN# editor makes it easy to design all the icons required by any program. Besides the usual drawing tools and icon design area, it displays the icons against both white and patterned backgrounds so that you can see how it will look on screen.

FIGURE 99. Application ICN# for SimpleBase.



From top to bottom you can use this single editor to create black and white ICN# and icn# resources, as well as 4- and 8-bit color icl4, ics4, icl8, and ics8 icon resources and the icon masks (used to separate the icon from the background pattern). All of these icon resources together are known collectively as an ICN# family.

The vers Resource

The vers resource contains the application version information. It's normally displayed when the user highlights the application and chooses **Get Info** from the **File** menu while in the *Finder*. An example of *SimplaBase* version information display is shown in Figure 100.

Setting the vers information is done in the vers editor shown in Figure 101. Here is where you set things like the application's version number, its stage of development (development, alpha, beta, and final), the country code (identifies the script system the version of the software was developed for), and provide short and long version strings for the **Get Info** window. Additionally, a vers resource should have a resource ID of 1 to specify the file version, or a resource ID of 2 to represent the version for a set of files.

FIGURE 100. Get Info display for SimpleBase.

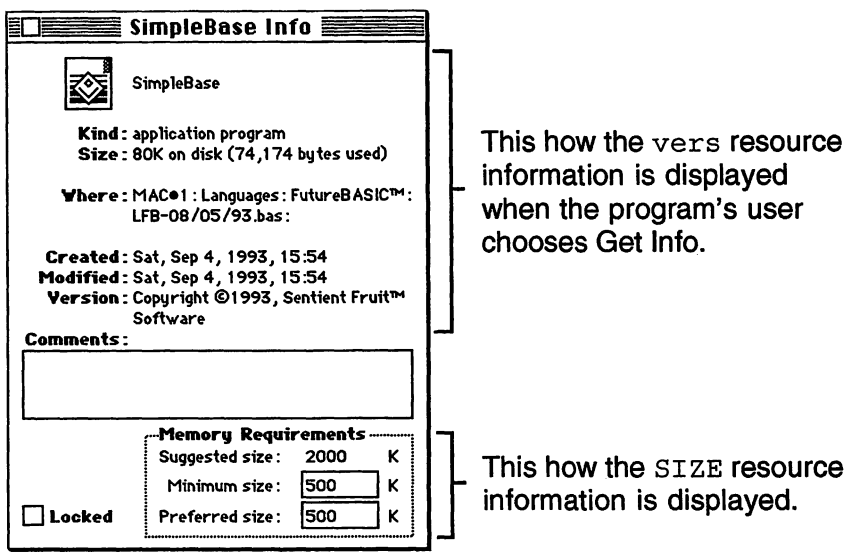
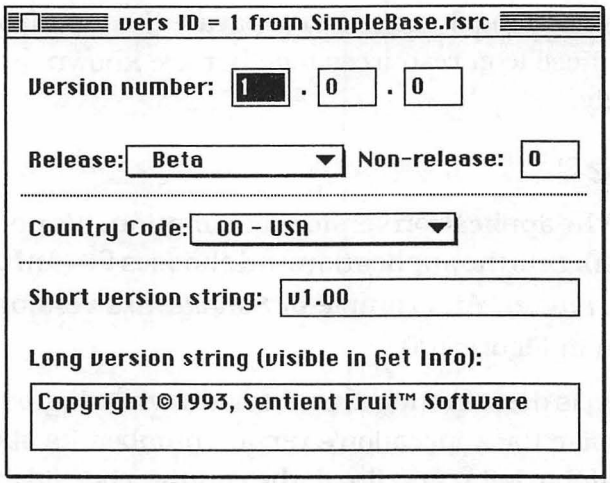


FIGURE 101. vers editor in ResEdit.



System 7 Resources

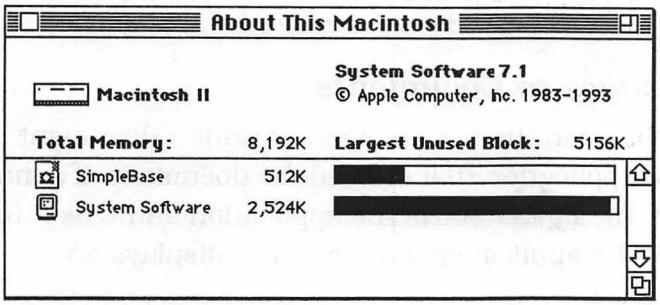
The following resources are required to provide additional support for System 7 (and MultiFinder) features. They should be added to all programs you write, regardless of which system the user will ultimately use your program on. User's of System 6 do use MultiFinder occasionally, and hopefully will upgrade to System 7 at some time. So help them out by making your application System 7 friendly right from the start.

The SIZE Resource

Because System 7 can run multiple applications at once, and because there is a finite amount of RAM memory in a particular machine, parcelling that memory out is vitally important. Since the System can't determine on its own what amount of memory an application might require, the SIZE resource was created. This enables the System to ask the application how much memory it requires. It then allocates enough space for the application, and launches it into the allocated memory space.

The SIZE resource under System 7 (and System 6 MultiFinder) describes both the upper and lower limits of memory space an application requires. The minimum memory setting is vitally important since it describes the least amount of memory an application can still operate in, albeit at a reduced capacity. In most cases the maximum memory setting describes the typical user requirements when working with the program. In some cases, if the user

FIGURE 102. Determining memory requirements.



is making unusually large demands on a program's memory space, the memory should be increased.

Additionally, the `SIZE` resource provides the operating system with additional information on whether the application is 32-bit clean, it can accept suspend and resume events, does it support stationary, and many more. We'll examine which ones should be set later in the chapter.

Determining Memory Requirements

One question I am often asked is how to determine the suggested, minimum, and preferred sizes for an application. While it's easy to just pick a size for each entry, doing so intelligently requires both some work and some thought on your part.

First, write and build your application. Then run it as hard as you can memory wise. Open as many files as the application allows. Build every array as large as required. Work the application as hard as you can. Periodically, examine the memory display in the *Finder* as shown in Figure 102. See how much memory the dialog shows the application actually requires.

In our example, *SimpleBase* is only using about 80K of its allocated 512K with no files open. Adding 20 or 25% of safety factor gives a total of about 100K for the minimum setting. However, when a file is open, it actually uses very little more, about 90K or so. This is because *SimpleBase* only keeps one record in memory at a time instead of the entire database. So, adding 25% to that (with some rounding up) I get a maximum memory setting of 128K[1].

Now, on both memory extremes, we've ensured that the program won't bump against a too small memory problem or request more memory than it can ever use and prevent other programs from running.

1. A nice even multiple of 2.

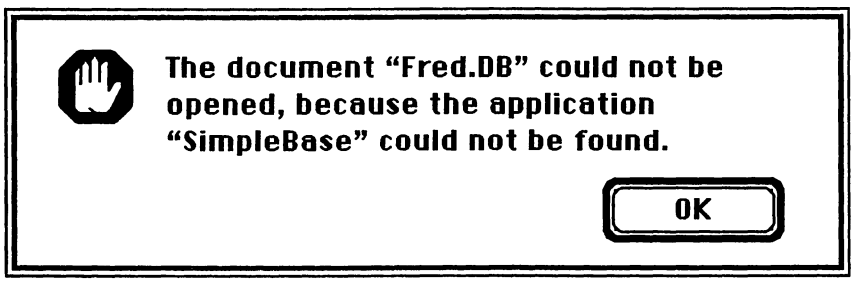
- *Note that if no `SIZE` resource is included in your application's resource file, FB will use a copy of its own `SIZE` resource which has a preferred setting of 2000K. Probably much larger than your program may ever need.*

Adding Finder Messages to Documents

Under *System 7*, when the user attempts to open or print a document the *Finder* will search for the application that created the document. If a match of signature types is found, the *Finder* starts the application and tells it to open or print the document. If the application isn't found, it displays an application-unavailable alert.

If the document is of the type `TEXT` or `PICT` and the *TeachText* application is available, the *Finder* will offer to open the document with *TeachText*. If *TeachText* is not available, the *Finder* displays an alert box like that shown in Figure 103.

FIGURE 103. Standard application-unavailable alert.

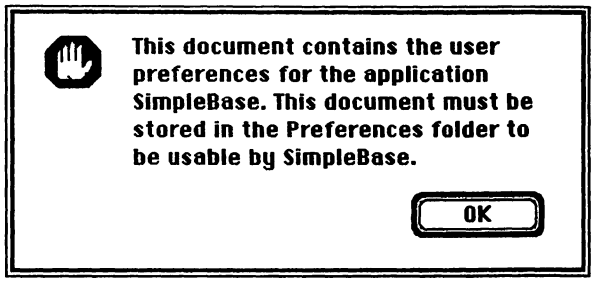


Before displaying the alert, the *Finder* searches the document for one of two custom `STR` resources. If the document is one that users can open, supply a `STR -16397` resource containing the application name. If the file is a preferences file, or one that is used by the application but one that users shouldn't open, supply a `STR -16396` resource containing a short message describing why the user can't open the file as shown for a fictional preferences file in Figure 113. Both resoures should be made purgeable.

Adding Balloon Help Resource

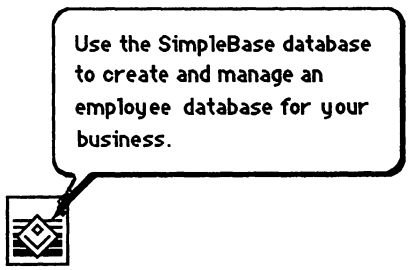
The *System 7 Finder* provides balloon help for online assistance of users. Whenever the user chooses **Show Balloons** from the (?) menu, descriptive messages appear inside of cartoon-style balloons as the users moves the cursor over an area of the screen (window, control, dialog) that has a help resource associated with it.

FIGURE 104. Custom application-unavailable alert.



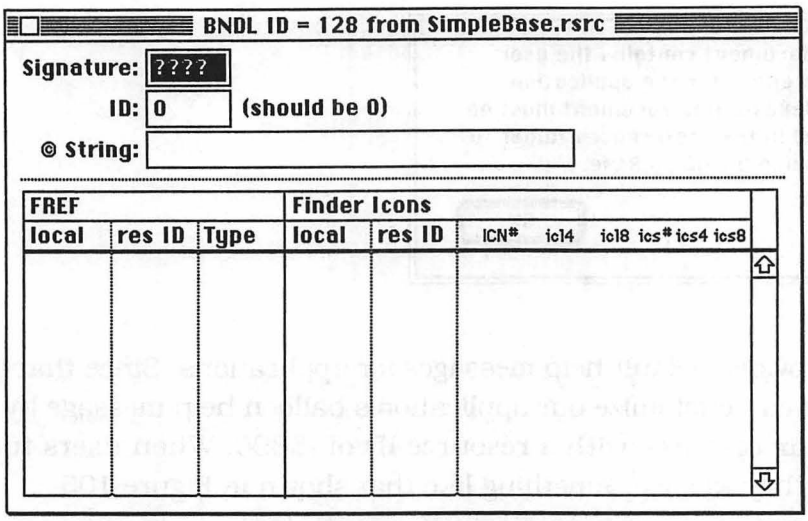
The *Finder* provides default help messages for applications. Since that isn't much fun, we can customize our application's balloon help message by creating a `hfdr` resource with a resource ID of -5696. When users turn on balloon help, they will see something like that shown in Figure 105.

FIGURE 105. Balloon help for SimpleBase.



- *Unfortunately, you can't overide the default document balloon help, just applications.*

FIGURE 106. Empty BNDL editor.



Regular Exercise

It's time to add all of the previously mentioned resources to the *SimpleBase.rsrc* file. Take it one step at a time and add them just as described.

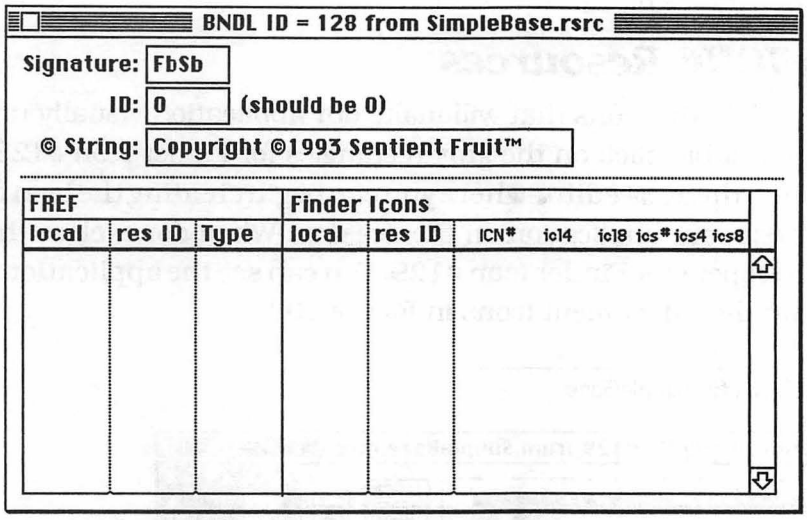
Creating BNDL Resources

From the BNDL editor it's possible to create the BNDL, FREF, ICN#, and signature resources. Start by running *ResEdit* and opening the *SimpleBase.rsrc* file. Choose **Create New Resource** from the **Resource** menu and click the BNDL resource type, then **OK**. *ResEdit* will create a blank BNDL resource like that shown in Figure 106. Finally, choose **Extended View** from the **BNDL** menu so that all of the resource bundled in the BNDL resource can be viewed.

Creating a Signature Resource

We start by adding our creator type to the BNDL editor. This will automatically generate a signature resource of the specified 4-character type within the resource file. We also add a small string to the resource. It's not required, and many applications leave the signature resource blank, but I like to place a copyright notice there. You can see the signature type and string in Figure 107 just as it was entered.

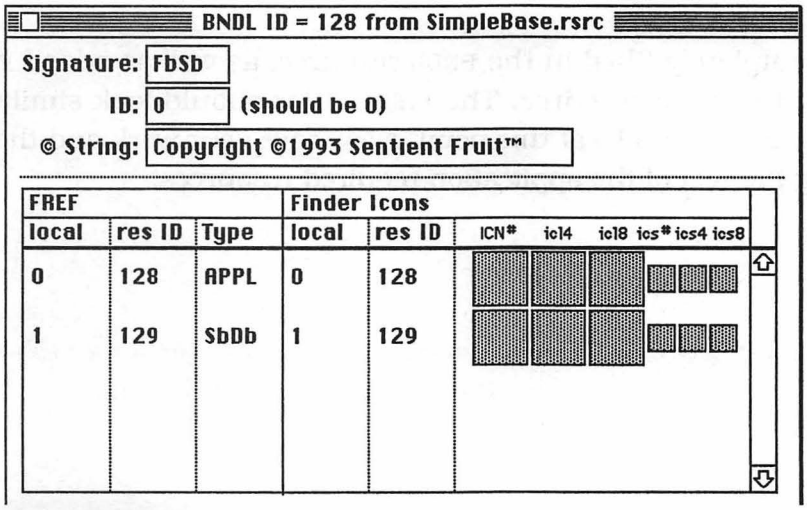
FIGURE 107. Creating a signature resource in the BNDL editor.



Creating FREF Resources

With the BNDL editor window frontmost, choose **Create New File Type** from the **Resources** menu. *ResEdit* will add a complete FREF/ICN# link. Enter the appropriate FREF types beginning with APPL. Also, add the resource ID to the ICN# containing the application icon. Choose **Create New File Type** again and add a FREF for our data files as SbDb and the resource ID to the file ICN#. When finished, it should look like the screen shot in Figure 108.

FIGURE 108. Adding FREFs to BNDL resource.

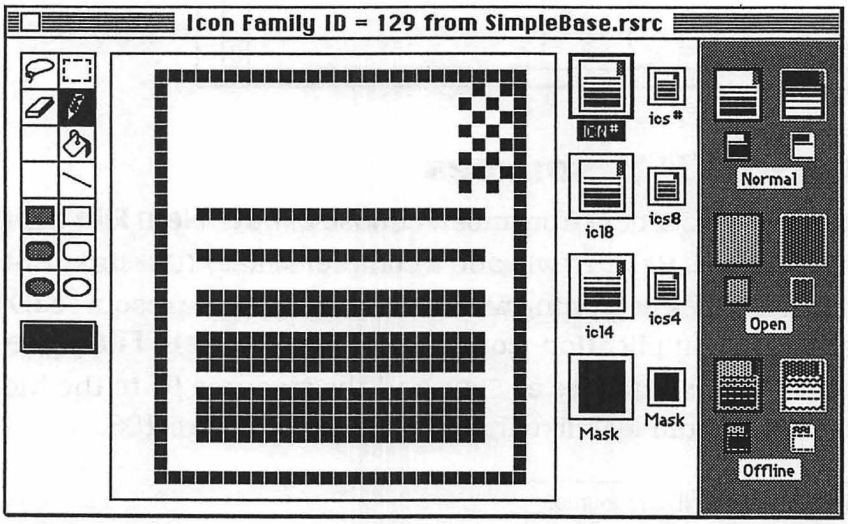


- *Note that you shouldn't change any of the Local ID settings. Just let ResEdit assign those as it sees fit.*

Creating ICN# Resources

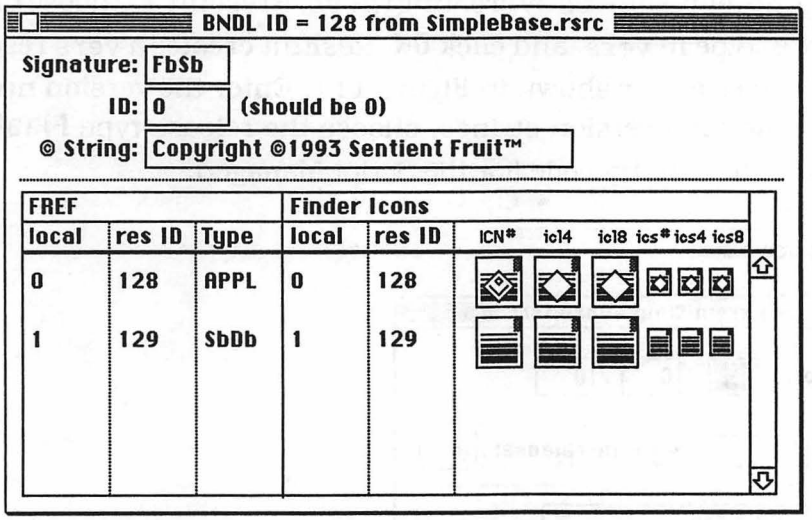
Next, it's time to add the icons that will make our application visually unique on the desktop. Double-click on the gray rectangles for Finder Icon #128. *ResEdit* will open the ICN# editor where you can begin creating the icon family that will represent your application on the desktop. When done, close the ICN# editor and repeat for Finder Icon #129. You can see the application icons in Figure 99 and their document icons in Figure 109.

FIGURE 109. Document icons for SimpleBase.



You've now completely filled in the BNDL resource, as well as added FREFs, ICN#s, and a signature resource. The BNDL editor should look similar to what is shown in Figure 110 at this point. Now, save your work and then begin creating the rest of the application required resources.

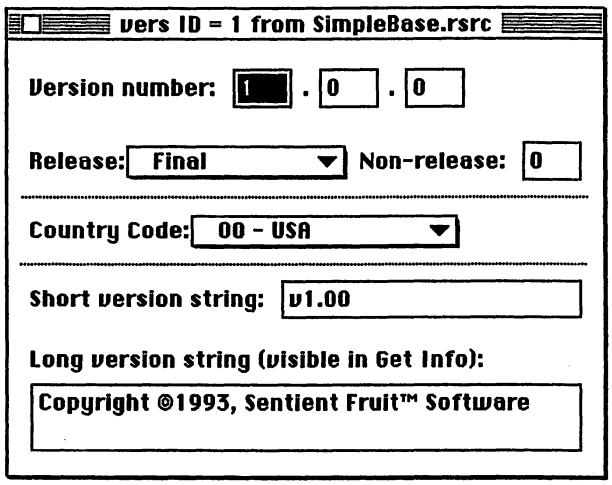
FIGURE 110. Complete BNDL for SimpleBase.



Creating a vers Resource

We'll begin the other resources by creating the vers resource. Choose **Create New Resource**, type in vers, and click **OK**. *ResEdit* creates a vers resource and opens the vers editor shown in Figure 111. Enter the version number (1.0.0), short and long version strings, choose the release type **Final**, and finish by setting the country code (for the Script Manager).

FIGURE 111. vers resource editor.



To complete the vers resource, choose **Get Resource Info** from the **Resources** menu and change the vers resource ID to 1. Save your changes and you're done with this resource.

Creating a SIZE Resource

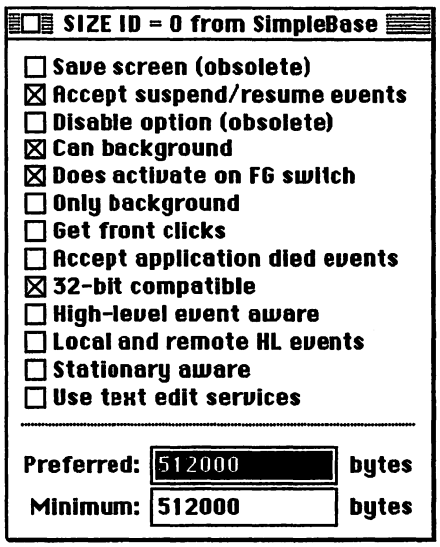
To create an SIZE resource, select **Create New Resource** from the **Resources** menu, find the SIZE type in the list. Click **OK**. When the SIZE editor appears, enter the suggested, minimum, and preferred application memory requirements, then click on the attributes your application requires. By default, you should always select the attibutes shown in Figure 112. Each of the selected attributes does the following (the others are not detailed here):

- *Accept suspend/resume events* – tells the operating system that your application can process suspend/resume events.
- *Can background* – when set, it tells the operating system that your application wants to receive null events while it's in the background.

- *Can activate on FG switch* – tells the operating system that your application should receive the mouse down and up events used to bring your application to the foreground.
- *32-bit compatible* – indicates that your program is 32-bit clean. Required for all new machines, and especially when running under System 7.

Finally, when everything is set as shown in Figure 112, use **Get Resource Info** to change the SIZE resource ID number to 0.

FIGURE 112. SIZE resource settings.

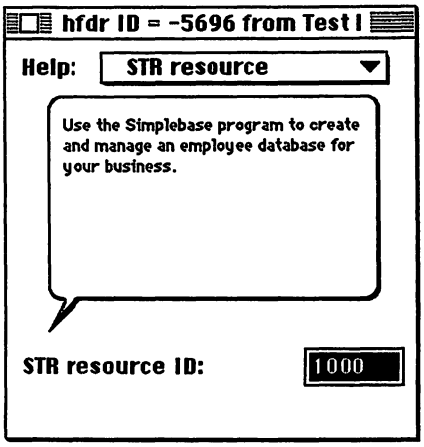


Creating the hfdr Resource

To create the hfdr resource, select **Create New Resource** from the **Resources** menu and find the hfdr type in the list. Click **OK**. When the hfdr editor appears, select STR format for the message from the Help popup list, then double-click on the balloon to access the STR editor. Enter the text that will appear when the user has balloon help on, then save your work.

Use **Get Resource Info** to change the hfdr resource ID number to -5696. For *SimpleBase*, the balloon help should appear as shown in Figure 113.

FIGURE 113. Custom help balloon for SimpleBase.



Cooldown

That's it for the system resources. In this chapter we learned all about the BNDL, FREF, ICN#, and signature resources. When properly linked, they provide the *Finder* with the capability to display your application icon and open your application when a user double-clicks on a program file.

Additionally, we learned about some other resources like SIZE, vers, hfdr and others.

Final Touches

Warm-up

This chapter will complete our work on *SimpleBase*. What's left are some final touches that you should add to *SimpleBase* and your own programs to make them friendlier to users.

Which System am I?

Right off the top you should know that your programs will not operate on all Macintosh computers, nor under all System versions (see the FB *Getting Started* manual, Minimum system requirements). So the first order of business when your program begins running, should always be to check that it can run under the host machine and operating system.

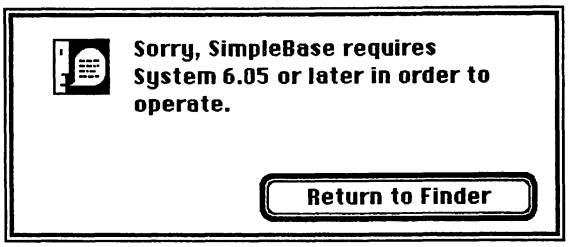
We can check our environment quite easily using the SYSTEM function. For example, to see if the program is running under a System version earlier than 6.05 we use the code shown in Program 159.

PROGRAM 159. Checking System version.

```
LONG IF SYSTEM (_sysVers) < 605
  item = FN ALERT (_versALRT, 0)
  END
END IF
```

Where the alert displayed is specific to the problem. In this case it should tell the user that the program cannot run under the current operating system. The ALRT resource added to the program might look like this.

FIGURE 114. Wrong system alert.



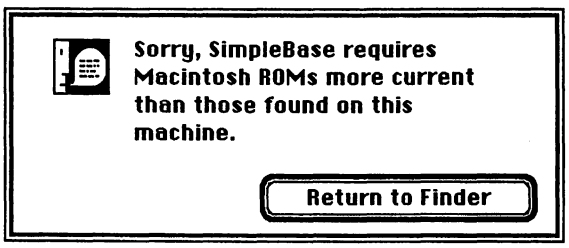
Additionally, we must always make sure that the program executes on is at least a MacPlus or newer. Another similar check should be made as shown in Program 160 which displays its own ALRT resource describing the particular problem to the user.

PROGRAM 160. Checking machine version.

```
LONG IF SYSTEM (_macType) < _envMacPlus
   item = FN ALERT (_machALRT, 0)
   END
END IF
```

In many cases a program can effectively get by using these two alert messages. Using *ResEdit*, add these two ALRT resources to *SimpleBase's* resource file. Next, let's open the *SimpleBase.glbl* file and add the two alert constants (_versionALRT and _machALRT), save and close. Finally, open the *SimpleBase.main* file and add the two testing routines to the Initialize function.

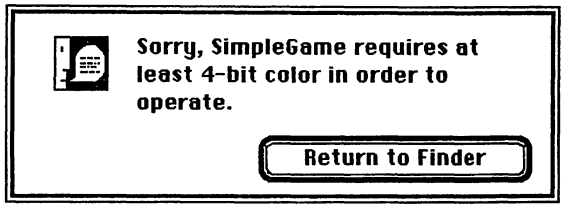
FIGURE 115. Sample wrong ROMs alert.



Screen Colors

Other programs may have different operating requirements. For example, a game program may require a color monitor before the user can play it. Or, more commonly, the user has a color monitor, but it's set to a different color

FIGURE 116. Incorrect color setting alert.



level than that required by the program. The user's monitor is set to black and white, but the game requires 4-bit color.

For this situation, a two-tiered test is required. We need to know first, is the program running on a color monitor, and secondly, if it is, is it set to the correct level for the program. The appropriate alert messages might look like Figure 116 while the code to implement this type of tiered color check appears in Program 161.

PROGRAM 161. Checking monitor color level

```
myMinColorLevel = 4
LONG IF SYSTEM (_maxColors) < myMinColorLevel
  item = FN ALERT (_noColorSpptALRT, 0)
  END
XELSE
  LONG IF SYSTEM (_crntDepth) < myMinColorLevel
    item = FN ALERT (_resetColorALRT, 0)
    END
  END IF
END IF
```

Opening and Printing File

Macintosh users are used to the operating system keeping track of a diversity of information. One particular bit that comes in very handy is the ability to double-click on a program's file and have the application launch, then open or print the selected file or files. Adding this functionality to your program is quickly done.

The statement that makes it all possible if `FINDERINFO. FINDERINFO` returns all information required by the program in order for it to open or print double-clicked files. This information includes a file count, filenames, file types, and volume reference numbers. The program can than examine each item in turn to determine whether it can open the particular file.

The key to using FINDERINFO is the proper organization of program routines for opening and printing files. If you adhere to the principles detailed in creating *SimpleBase* it becomes an easy matter to add this type *Finder* support.

We start by defining two new constants in the *SimpleBase.glbl* file:

```
_openFiles = 0
_printFiles = 1
```

These are used to identify which operation the *Finder* wants us to do when it passes us a file. The CheckIncomingFiles routine shown in Program 162 shows how to extract the infomration we need to successfully open or print a file.

PROGRAM 162. Adding Finder support.

```
LOCAL FN CheckIncomingFiles
  maxFiles% = 1
  doWhat% = FINDERINFO (maxFiles%, gFileName$, fileType&, gWDRefNum%)
  LONG IF (maxFiles% > 0) AND (fileType& = _"SbDb")
    SELECT doWhat%
      CASE _openFiles
        FN ItemOpen
      CASE _printFiles
        FN ItemOpen
        FN ItemPrint
    END SELECT
  END IF
END FN
```

We call this routine last in the Initialize routine during program start-up. It uses FINDERINFO to determine if any filenames have been passed from the *Finder*. If no files were passed to the program, it returns without doing a thing.

If files have been passed, it examines each file in turn and processes it appropriately. The routine starts by specifying the maximum number of files the program will accept in the variable maxFiles%. maxFiles% is used by the FINDERINFO function to examine the file list from the *Finder*. When done, maxFiles% contains the actual number of files passed to the program.

This setting of maxFiles% is critical to successfully using FINDERINFO to open or print files. If maxFiles% is not set (or ever equates to zero) your program will be unable to open or print files during startup.

Once FINDERINFO has been called, the routine examines maxFiles%. If maxFiles% doesn't equal zero further processing takes place. It then examines the doWhat% variable to determine exactly how to respond. doWhat% will contain a zero if the file should be opened, and 1 if it should be printed.

Handling Multiple File Types

Now that we know what to do with double-clicked files, let's examine how to add another filter to the file evaluation, that of file types. If Simplebase had a preferences file, we may want the program to open, but not display the preferences directly.:

```
DEF OPEN "FREDfred"
```

This statement specified the data file's type and creator. The type being a 4-character code that uniquely identifies the file and the creator being another 4-letter code that identifies the application that created the file. The *Finder* uses these bits of information to open the file's creator application when its double-clicked.

We require this bit of information if our program can open more than one file type. We first examine the file type to determine if, one, the file type is actually one our program can open, and second, if it is one we can open, then direct it to the correct opening subroutine. A SELECT/END SELECT structure works best for this type of filtering. That way, if we ever need to add an additional file type, just add another CASE to the list for the appropriate file type. The file type filtering routine might look like this:

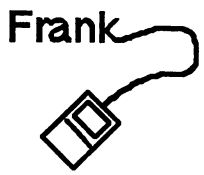
```
SELECT type&
  CASE _"lora"
    FN OpenLoraTypeFile
  CASE _"todd"
    FN OpenToddTypeFile
  CASE _"john"
    FN OpenJohnTypeFile
  CASE ELSE
END SELECT
```

Cooldown

Well, that's it. We're done with *SimpleBase*. That's not to say that you have to be done with it. You probably already have some great ideas bubbling around in your head on improving *SimpleBase* or writing other programs.

In this book, I've tried to give you the knowledge required to write applications on the Macintosh. Along the way, we've seen how to implement common features of the Macintosh interface, as well as explained the principles behind their presence and reasons for their actions. We've also covered many useful programming techniques that are guaranteed to make your other projects easier.

Like my favorite cowboy always said, "Happy coding to you".



PS:

Now that I feel like I'm on some kind of a product roll, watch for:

Learning FutureBASIC: Toolbox 1

Learning FutureBASIC: Macintosh BASIC Power Video

Bibliography

The following references may help your understanding when using this book, I know they helped me.

Books...

Hogan, Thom. "The Programmer's Apple Mac Sourcebook" Microsoft Press.
Chernicoff, Stephen. "Macintosh Revealed Volumes 1-4" Hayden Books.
Gariepy, Michael. "FutureBASIC Handbook" Zedcor, Inc.
Gariepy, Michael. "Programming the Macintosh with FutureBASIC" Zedcor, Inc.
"Inside Macintosh, Volumes I-VI" Addison-Wesley.
"Inside Macintosh 2nd Ed: Files" Addison-Wesley.
"Inside Macintosh 2nd Ed: Overview" Addison-Wesley.
"Inside Macintosh 2nd Ed: Text" Addison-Wesley.
"Inside Macintosh 2nd Ed: Macintosh Toolbox Essentials" Addison-Wesley.
Knaster, Scott. "Macintosh Programming Secrets 2nd Ed"Addison-Wesley.
Little, Gary and Swihart, Tim. "Programming for System 7" Addison-Wesley.
Turovich, L. Frank. "FutureBASIC Reference" Zedcor, Inc.

Magazines...

Inside Basic magazine, *The Journal of Macintosh BASIC Programming*
Ariel Publishing, Inc.
PO Box 398
Pateros, WA 98846-0398
Phone: 509.923.2249
America Online: Ariel

MacTech magazine, *The Macintosh Programming Journal*
PO Box 250055
Los Angeles, CA 90025-0055
America Online: MacTechMag

develop, *The Apple Technical Journal*
Apple Computer, Inc.
PO Box 531
Mt. Morris, IL 61054
AppleLink: DEV.SUBS

Appendix

SimpleBase.glbl

```
' --- CONSTANTS ----------------------------------
' >>> FILE ID
_dbFileID       = 1
' >>> WINDOWS
_dbEntryWIND    = 1
_dbFindWIND     = 2
_aboutWIND      = 3
_helpWIND       = 4
_printWIND      = 5
_gotoWIND       = 6

' >>> BUTTONS
' >>> DATA ENTRY WINDOW
_newRecBTN      = 1
_firstRecBTN    = 2
_prevRecBTN     = 3
_nextRecBTN     = 4
_lastRecBTN     = 5
_findRecBTN     = 6
_clearRecBTN    = 7
_programBTN     = 8
_marketBTN      = 9
_officeBTN      = 10

' >>> FIND WINDOW
_findBTN        = 1
_cancelBTN      = 2
_ignoreCaseBTN = 3

' >>> ABOUT WINDOW
_okBTN          = 1

' >>> HELP WINDOW
_helpSCROLL     = 1
_prevHelpBTN    = 2
_nextHelpBTN    = 3

' >>> PRINT WINDOW
_printBTN       = 1
_thisRecBTN     = 3
_allRecBTN      = 4
_selectRecBTN   = 5

' >>> GOTO WINDOW
_gotoBTN        = 1

' >>> EDIT/PICTURE FIELDS
' >>> FIND WINDOW
_dbFindFLD      = 1

' >>> HELP WINDOW
_helpFLD        = 1

' >>> PRINT WINDOW
_firstPrFLD     = 1
_lastPrFLD      = 2

' >>> DATA ENTRY WINDOW
_dbNameFLD      = 1
```

```
_dbAddrFLD      = 2
_dbCityFLD      = 3
_dbStateFLD     = 4
_dbZipFLD       = 5
_dbPhoneFLD     = 6
_dbFaxFLD       = 7
_dbPhotoFLD     = 8
_recordFLD      = 50

' >>> GOTO WINDOW
_gotoFLD        = 1

' >>> APPLE MENU
_iAbout      = 1
_iHelp       = 2

' >>> FILE MENU
_mFile       = 1
_iNew        = 1
_iOpen       = 2
_iClose      = 3
' -------------
_iPageSetup = 5
_iPrint      = 6
' -------------
_iQuit       = 8
' >>> EDIT MENU
_mEdit       = 2
_iUndo       = 1
' -------------
_iCut        = 3
_iCopy       = 4
_iPaste      = 5
_iClear      = 6
' -------------
_iSelectAll = 8
' -------------
_iCopyRec    = 10
_iPasteRec   = 11

' >>> RECORD MENU
_mRecord     = 3
_iFirstRec   = 1
_iPrevRec    = 2
_iNextRec    = 3
_iLastRec    = 4
' -------------
_iFindRec    = 6
_iGotoRec    = 7
' -------------
_iClearRec   = 9

' >>> ALRT IDs
_aboutALRT     = 128
_tooLongALRT   = 129
_noFindALRT    = 130
_machErrALRT   = 131
_sysErrALRT    = 132
_notRecordALRT= 133

' >>> STRING IDs
_windowSTR     = 1000
_buttonSTR     = 2000
_fieldSTR      = 3000

' >>> HELP ID
_minHelpID     = 1001
_maxHelpID     = 1009

' >>> MISC STUFF
```

```
_gutterAdj     = 30
_openFiles     = 0
_printFiles    = 1
_tab           = 9
_cr            = 13

' --- RECORD STRUCTURES -------------------------

DIM RECORD dbRecord
  DIM dbName$;64
  DIM dbAddr$;64
  DIM dbCity$;32
  DIM dbMyState$;4
  DIM dbZip$;12
  DIM dbPhone$;12
  DIM dbFax$;12
  DIM dbDeptNum%
  DIM dbPictID%
  DIM dbExtra&
DIM END RECORD .dbRecordSize

' --- VARIABLES ---------------------------------

' >>> FILE
DIM gOpenRecord%, gMaxRecInFile%
DIM gWDRefNum%
DIM 255 gFileName$

' >>> HELP WND
DIM gHelpID%

' >>> PRINT WND
DIM gPrFirstRec%
DIM gPrLastRec%
DIM gPrintFlag%

' >>> RECORD
DIM gEmployee.dbRecordSize
DIM gPictH&

' >>> FIND WND
DIM gCaseFlag%
DIM 127 gSearch$

' >>> PROGRAM GLOBALS
DIM gQuit

' --- GLOBAL PROJECT FUNCTIONS ----------------------

' --- Project.Incl
DIM gCursorPtr&
DIM gEFClickPtr&
DIM gTabEventsPtr&
DIM gCheckBoxPtr&
DIM gRadioBtnPtr&
DIM gHiliteBtnPtr&
DIM gReturnToBtnPtr&

' --- EditMenu.Incl
DIM gEditMenuPtr&

' --- Printing.Incl
DIM gDoPrintPtr&
```

Project.Incl

```
' --- HEADER ------------------------------------

INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels

' --- CONSTANTS ---------------------------------

GLOBALS "SimpleBase.glbl"
END GLOBALS

' --- DIALOGEVENT.INCL
DEF FN CursorHandler (cursEvtID%,dlgID%) USING gCursorPtr&
DEF FN EFClickEvent (fieldID%) USING gEFClickPtr&
DEF FN TabShiftTabEvents (dlgEvnt%, startFld%, lastFld%) USING gTabEventsPtr&
DEF FN CheckBoxHandler% (btnID%) USING gCheckBoxPtr&
DEF FN RadioBtnHandler% (lowBtnID%, highBtnID%, setBtnID%) USING gRadioBtnPtr&
DEF FN HiliteSelectedButton (btnID%) USING gHiliteBtnPtr&
DEF FN ChangeReturnToBtn (@evntIDPtr&, btnID%) USING gReturnToBtnPtr&
' --- PRINTING.INCL

DEF FN DoPrinting (readRecPtr&) USING gDoPrintPtr&
```

SimpleBase.Incl

```
' --- HEADER ------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels

' --- CONSTANTS ---------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS

' --- INCLUDES ----------------------------------
INCLUDE "Project.Incl"

' --- FUNCTIONS ---------------------------------
' === MISC MENU FUNCTIONS ===

LOCAL FN UpdateMenus
  SELECT WINDOW (_outputWClass)
    CASE _dbEntryWIND
      MENU _mFile  , 0, _enable
      MENU _mEdit  , 0, _enable
      MENU _mRecord, 0, _enable
      MENU _mFile, _iNew      , _enable, "New Record"
      MENU _mFile, _iOpen     , _disable
      MENU _mFile, _iClose    , _enable
      MENU _mFile, _iPageSetup, _enable
      MENU _mFile, _iPrint    , _enable
      MENU _mEdit, _iSelectAll, _enable
      MENU _mEdit, _iCopyRec  , _enable
      MENU _mEdit, _iPasteRec , _enable
    CASE _dbFindWIND, _aboutWIND, _helpWIND, _gotoWIND
      MENU _mFile  , 0, _disable
      MENU _mEdit  , 0, _disable
      MENU _mRecord, 0, _disable
    CASE ELSE
      MENU _mFile  , 0, _enable
      MENU _mEdit  , 0, _enable
      MENU _mRecord, 0, _disable
      MENU _mFile, _iNew       , _enable, "New"
      MENU _mFile, _iOpen      , _enable
      MENU _mFile, _iClose     , _disable
      MENU _mFile, _iPageSetup , _disable
```

```
      MENU _mFile, _iPrint     , _disable
      MENU _mEdit, _iSelectAll , _disable
      MENU _mEdit, _iCopyRec   , _disable
      MENU _mEdit, _iPasteRec  , _disable
  END SELECT
END FN

' === MISC RECORD FUNCTIONS ===

LOCAL FN SaveEmployeeGraphic
  resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, _fsRdWrPerm)
  LONG IF resRef% <> _nil
    LONG IF gEmployee.dbPictID% > _nil
      pictH& = FN GETPICTURE (gEmployee.dbPictID%)
      LONG IF (pictH& <> _nil) AND (FN RESERROR = _noErr)
        CALL RMVERESOURCE (pictH&)
      END IF
      CALL ADDRESOURCE (gPictH&, _"PICT", gEmployee.dbPictID%, gEmployee.dbName$)
      CALL SETRESATTRS (gPictH&, _resPurgeable%)
      CALL CHANGEDRESOURCE (gPictH&)
      CALL WRITERESOURCE (gPictH&)
      CALL DETACHRESOURCE (gPictH&)
    END IF
    CALL CLOSERESFILE (resRef%)
  END IF
END FN

LOCAL FN ReadEmployeeGraphic
  resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, _fsCurPerm)
  LONG IF resRef% <> _nil
    LONG IF gEmployee.dbPictID% > _nil
      DEF DISPOSEH (gPictH&)
      gPictH& = FN GETPICTURE (gEmployee.dbPictID%)
      LONG IF (gPictH& <> _nil) AND (FN RESERROR = _noErr)
        EDIT$ (_dbPhotoFLD) = &gPictH&
        CALL DETACHRESOURCE (gPictH&)
      END IF
    END IF
    CALL CLOSERESFILE (resRef%)
  END IF
END FN

LOCAL FN CheckFieldLength$ (fieldID%, maxLen%)
  tmp$ = EDIT$(fieldID%)
  LONG IF LEN (tmp$) > maxLen%
    item% = FN ALERT (_tooLongALRT, 0)
    tmp$ = LEFT$ (tmp$, maxLen%)
  END IF
END FN = tmp$

LOCAL FN EFtoRecordField
  oldWnd% = WINDOW (_outputWnd)
  WINDOW OUTPUT #_dbEntryWIND
  gEmployee.dbName$    = FN CheckFieldLength$ (_dbNameFLD, 63)
  gEmployee.dbAddr$    = FN CheckFieldLength$ (_dbAddrFLD, 63)
  gEmployee.dbCity$    = FN CheckFieldLength$ (_dbCityFLD, 31)
  gEmployee.dbMyState$ = FN CheckFieldLength$ (_dbStateFLD, 3)
  gEmployee.dbZip$     = FN CheckFieldLength$ (_dbZipFLD, 11)
  gEmployee.dbPhone$   = FN CheckFieldLength$ (_dbPhoneFLD, 11)
  gEmployee.dbFax$     = FN CheckFieldLength$ (_dbFaxFLD, 11)
  FN SaveEmployeeGraphic
  WINDOW OUTPUT #oldWnd%
END FN

LOCAL FN RecordFieldToEF
  oldWnd% = WINDOW (_outputWnd)
  WINDOW OUTPUT #_dbEntryWIND
  EDIT$(_dbPhotoFLD)      = ""
  EDIT$ (_dbNameFLD)      = gEmployee.dbName$
  EDIT$ (_dbAddrFLD)      = gEmployee.dbAddr$
```

```
  EDIT$ (_dbCityFLD)      = gEmployee.dbCity$
  EDIT$ (_dbStateFLD)     = gEmployee.dbMyState$
  EDIT$ (_dbZipFLD)       = gEmployee.dbZip$
  EDIT$ (_dbPhoneFLD)     = gEmployee.dbPhone$
  EDIT$ (_dbFaxFLD)       = gEmployee.dbFax$
  FN RadioBtnHandler% (_programBTN, _officeBTN, gEmployee.dbDeptNum%)
  FN ReadEmployeeGraphic
  ' ••• display record and file data in window
  tmp$ = STR$ (gOpenRecord%) + " of" + STR$(gMaxRecInFile%) + " records"
  EDIT$ (_recordFLD) = tmp$
  WINDOW OUTPUT #oldWnd%
END FN

' === MISC FILE HANDLING

CLEAR LOCAL
LOCAL FN DBWriteRecord
  DEF OPEN "SbDbFbSb"
  OPEN "R", #_dbFileID, gFileName$, _dbRecordSize, gWDRefNum%
  RECORD #_dbFileID, gOpenRecord%
  WRITE #_dbFileID, gEmployee
  gMaxRecInFile% = LOF (_dbFileID, _dbRecordSize) - 1
  CLOSE #_dbFileID
END FN

CLEAR LOCAL
LOCAL FN DBReadRecord
  DEF OPEN "SbDbFbSb"
  OPEN "R", #_dbFileID, gFileName$, _dbRecordSize, gWDRefNum%
  RECORD #_dbFileID, gOpenRecord%
  READ #_dbFileID, gEmployee
  gMaxRecInFile% = LOF (_dbFileID, _dbRecordSize) - 1
  CLOSE #_dbFileID
END FN

CLEAR LOCAL
LOCAL FN DBBlankRecord
  DEF BLOCKFILL (@gEmployee, _dbRecordSize, 0)    'make sure record is empty
END FN

CLEAR LOCAL
LOCAL FN DBNewEmployeeFile
  FN DBBlankRecord
  DEF OPEN "SbDbFbSb"
  OPEN "R", #1, gFileName$, , gWDRefNum%
  CLOSE #1
  CALL CREATERESFILE (gFileName$)
  gOpenRecord% = 0
  gEmployee.dbName$ = "Created by SimpleBase, from the book:"
  gEmployee.dbAddr$ = "Learning FutureBASIC: Macintosh BASIC Power"
  gEmployee.dbCity$ = "By Sentient Fruit™"
  FN DBWriteRecord
  INC (gOpenRecord%)
END FN

CLEAR LOCAL
LOCAL FN DBFindRecord
  originalRecNum% = gOpenRecord%
  IF gCaseFlag = _markedBtn THEN gSearch$ = UCASE$ (gSearch$)
  CURSOR _watchCursor
  gOpenRecord% = 1
  DO
    FN DBReadRecord
    test$ = gEmployee.dbName$
    IF gCaseFlag = _markedBtn THEN test$ = UCASE$ (test$)
    found% = INSTR (1, test$, gSearch$)
    INC (gOpenRecord%)
  UNTIL (found% <> 0) OR (gOpenRecord% > gMaxRecInFile%)
  CURSOR _arrowCursor
  LONG IF found% = 0
```

```
    CALL PARAMTEXT (gSearch$, gFileName$, "", "")
    item% = FN ALERT (_noFindALRT, 0)
    gOpenRecord% = originalRecNum%
    FN DBReadRecord
  END IF
  WINDOW #_dbEntryWIND
  FN RecordFieldToEF
END FN

LOCAL FN CheckRange (current%, minRange%, maxRange%)
  IF current% < minRange% THEN current% = minRange%
  IF current% > maxRange% THEN current% = maxRange%
END FN = current%

' === WINDOW FUNCTIONS ===

LOCAL FN WindowCapture (wndID%)
  closeFlag% = _true
  SELECT wndID%
    CASE _dbEntryWIND
      FN EFtoRecordField
      FN DBWriteRecord
    CASE _dbFindWIND
      tmp$ = EDIT$(_dbFindFLD)
      LONG IF LEN (tmp$) > 127                    'make sure find string doesn't
        tmp$ = LEFT$ (tmp$, 127)                  'exceed max gSearch$ length
      END IF
      gSearch$ = tmp$
    CASE _aboutWIND
    CASE _helpWIND
    CASE _printWIND
      gPrLastRec% = VAL (EDIT$ (_lastPrFLD))
      gPrLastRec% = FN CheckRange (gPrLastRec%, 1, gMaxRecInFile%)
      gPrFirstRec% = VAL (EDIT$ (_firstPrFLD))
      gPrFirstRec% = FN CheckRange (gPrFirstRec%, 1, gPrLastRec%)
    CASE _gotoWIND
      gOpenRecord% = VAL (EDIT$ (_gotoFLD))
      gOpenRecord% = FN CheckRange (gOpenRecord%, 1, gMaxRecInFile%)
  END SELECT
END FN = closeFlag%

LOCAL FN WindowClose (wndID%)
  LONG IF FN WindowCapture (wndID%)
    SELECT wndID%
      CASE _dbEntryWIND
      CASE _dbFindWIND
      CASE _aboutWIND
      CASE _helpWIND
      CASE _printWIND
      CASE _gotoWIND
    END SELECT
    WINDOW CLOSE #wndID%
  END IF
END FN

LOCAL FN BuildEntryWindow
  tmp$ = STR#(_windowSTR, _dbEntryWIND)
  WINDOW #-_dbEntryWIND, tmp$, (0,0)-(500,290), _docNoGrow, _dbEntryWIND
  TEXT _sysFont, 12
  ' ••• BUTTONS
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _newRecBTN)
  BUTTON _newRecBTN, _activeBtn, tmp$, (380,20)-(480,40)  , _shadow
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _firstRecBTN)
  BUTTON _firstRecBTN, _activeBtn, tmp$, (380,50)-(480,70)  , _push
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _prevRecBTN)
  BUTTON _prevRecBTN, _activeBtn, tmp$, (380,80)-(480,100)  , _push
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _nextRecBTN)
  BUTTON _nextRecBTN, _activeBtn, tmp$, (380,110)-(480,130), _push
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _lastRecBTN)
  BUTTON _lastRecBTN, _activeBtn, tmp$, (380,140)-(480,160), _push
```

```
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _findRecBTN)
  BUTTON _findRecBTN, _activeBtn, tmp$, (380,170)-(480,190), _push
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _clearRecBTN)
  BUTTON _clearRecBTN, _activeBtn, tmp$, (380,210)-(480,230), _push
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _programBTN)
  BUTTON _programBTN, _activeBtn, tmp$, (90,222)-(200,237), _radio
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _marketBTN)
  BUTTON _marketBTN, _activeBtn, tmp$, (90,238)-(200,253), _radio
  tmp$ = STR#(_dbEntryWIND_buttonSTR, _officeBTN)
  BUTTON _officeBTN, _activeBtn, tmp$, (90,254)-(200,269), _radio
  ' ••• INACTIVE EDIT/PICT FIELDS
  xPos = 85
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbNameFLD)
  EDIT FIELD #100, tmp$, (20,40)-(xPos-5,56), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbAddrFLD)
  EDIT FIELD #101, tmp$, (20,66)-(xPos-5,82), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbCityFLD)
  EDIT FIELD #102, tmp$, (20,92)-(xPos-5,108), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbStateFLD)
  EDIT FIELD #103, tmp$, (20,118)-(xPos-5,134), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbZipFLD)
  EDIT FIELD #104, tmp$, (20,144)-(xPos-5,160), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbPhoneFLD)
  EDIT FIELD #105, tmp$, (20,170)-(xPos-5,186), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbFaxFLD)
  EDIT FIELD #106, tmp$, (20,196)-(xPos-5,212), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbPhotoFLD)
  EDIT FIELD #107, tmp$, (20,222)-(xPos-5,238), _StatNoframed, _rightJust
  tmp$ = STR#(_dbEntryWIND_fieldSTR, _dbPhotoFLD + 1)
  EDIT FIELD #108, tmp$, (20,14)-(xPos-5,30), _StatNoframed, _rightJust
  EDIT FIELD #_recordFLD, "", (xPos,14)-(345,30), _StatNoframed, _leftJust
  ' ••• ACTIVE EDIT/PICT FIELDS
  EDIT FIELD #_dbNameFLD, "", (xPos,40)-(345,56), _framedNoCR, _leftJust
  EDIT FIELD #_dbAddrFLD, "", (xPos,66)-(345,82), _framedNoCR, _leftJust
  EDIT FIELD #_dbCityFLD, "", (xPos,92)-(345,108), _framedNoCR, _leftJust
  EDIT FIELD #_dbStateFLD, "", (xPos,118)-(170,134), _framedNoCR, _leftJust
  EDIT FIELD #_dbZipFLD, "", (xPos,144)-(200,160), _framedNoCR, _leftJust
  EDIT FIELD #_dbPhoneFLD, "", (xPos,170)-(200,186), _framedNoCR, _leftJust
  EDIT FIELD #_dbFaxFLD, "", (xPos,196)-(200,212), _framedNoCR, _leftJust
  PICTURE FIELD #_dbPhotoFLD, "", (215,118)-(345,270), _framedNoCR, _cropPict
  ' ••• SET WINDOW BUTTONS/ACTIVE FIELD
  EDIT FIELD #_dbNameFLD
END FN

LOCAL FN BuildFindWindow
  tmp$ = STR#(_windowSTR, _dbFindWIND)
  WINDOW #-_dbFindWIND, tmp$, (0,0)-(340,80), _docNoGrow_noGoAway, _dbFindWIND
  TEXT _sysFont, 12
  ' ••• BUTTONS
  tmp$ = STR#(_dbFindWIND_buttonSTR, _findBTN)
  BUTTON _findBTN, _activeBtn, tmp$, (250,50)-(320,70), _shadow
  tmp$ = STR#(_dbFindWIND_buttonSTR, _cancelBTN)
  BUTTON _cancelBTN, _activeBtn, tmp$, (160,50)-(230,70), _push
  tmp$ = STR#(_dbFindWIND_buttonSTR, _ignoreCaseBTN)
  BUTTON _ignoreCaseBTN, _activeBtn, tmp$, (20,50)-(150,70) , _checkBox
  ' ••• INACTIVE EDIT/PICT FIELDS
  tmp$ = STR#(_dbFindWIND_fieldSTR, 1)
  EDIT FIELD #100, tmp$, (15,15)-(50,30), _StatNoframed, _rightJust
  ' ••• ACTIVE EDIT/PICT FIELDS
  EDIT FIELD #_dbFindFLD, gSearch$, (55,15)-(320,30), _framedNoCR, _leftJust
END FN

LOCAL FN BuildAboutWindow
  item% = FN ALERT (_aboutALRT, 0)
END FN

LOCAL FN BuildHelpWindow
  tmp$ = STR#(_windowSTR, _helpWIND)
  WINDOW #-_helpWIND, tmp$, (0,0)-(400,260), _docZoom, _helpWIND
  TEXT _sysFont, 12
```

```
  ' ••• BUTTONS
  tmp$ = STR#(_helpWIND_buttonSTR, _prevHelpBTN)
  BUTTON #_prevHelpBTN, 1, tmp$, (20,5)-(100,25), _push
  tmp$ = STR#(_helpWIND_buttonSTR, _nextHelpBTN)
  BUTTON #_nextHelpBTN, 1, tmp$, (120,5)-(200,25), _push
  ' ••• INACTIVE EDIT/PICT FIELDS
  wndX = WINDOW (_width) : wndY = WINDOW (_height)
  EDIT FIELD #-_helpFLD, %gHelpID%, (4,34)-(wndX-4,wndY-4), _statNoframed, _leftJust
  SCROLL BUTTON #-_helpSCROLL,1,1,1,10, (wndX-16,31)-(wndX,wndY),_scrollVert
END FN

LOCAL FN BuildPrintWindow
  gPrintFlag% = _thisRecBTN
  tmp$ = STR#(_windowSTR, _printWIND)
  WINDOW #-_printWIND, tmp$, (0,0)-(300,125), _docNoGrow_noGoAway, _printWIND
  TEXT _sysFont, 12
  ' ••• BUTTONS
  tmp$ = STR#(_printWIND_buttonSTR, _okBTN)
  BUTTON _okBTN, _activeBtn, tmp$, (200,90)-(280,110)  , _shadow
  tmp$ = STR#(_printWIND_buttonSTR, _cancelBTN)
  BUTTON _cancelBTN, _activeBtn, tmp$, (100,90)-(180,110)  , _push
  tmp$ = STR#(_printWIND_buttonSTR, _thisRecBTN)
  BUTTON _thisRecBTN, 2, tmp$, (20,10)-(200,25), _radio
  tmp$ = STR#(_printWIND_buttonSTR, _allRecBTN)
  BUTTON _allRecBTN, 1, tmp$, (20,30)-(200,45), _radio
  tmp$ = STR#(_printWIND_buttonSTR, _selectRecBTN)
  BUTTON _selectRecBTN, 1, tmp$, (20,50)-(160,65), _radio
  ' ••• INACTIVE EDIT/PICT FIELDS
  tmp$ = "to"
  EDIT FIELD #100, tmp$, (205,50)-(235,65), _statNoframed, _centerJust
  ' ••• ACTIVE EDIT/PICT FIELDS
  tmp$ = STR$(gMaxRecInFile%)
  EDIT FIELD #_lastPrFLD, tmp$, (240,50)-(275,65), _framedNoCR, _centerJust
  EDIT FIELD #_firstPrFLD, "1", (165,50)-(200,65), _framedNoCR, _centerJust
END FN

LOCAL FN BuildGotoWindow
  tmp$ = STR#(_windowSTR, _gotoWIND)
  WINDOW #-_gotoWIND, tmp$, (0,0)-(200,80), _docNoGrow_noGoAway, _gotoWIND
  TEXT _sysFont, 12
  ' ••• BUTTONS
  tmp$ = STR#(_gotoWIND_buttonSTR, _gotoBTN)
  BUTTON _gotoBTN, _activeBtn, tmp$, (120,45)-(180,65), _shadow
  tmp$ = STR#(_gotoWIND_buttonSTR, _cancelBTN)
  BUTTON _cancelBTN, _activeBtn, tmp$, (20,45)-(80,65), _push
  ' ••• INACTIVE EDIT/PICT FIELDS
  tmp$ = STR#(_gotoWIND_fieldSTR, 1)
  EDIT FIELD #100, tmp$, (10,15)-(105,30), _StatNoframed, _rightJust
  ' ••• ACTIVE EDIT/PICT FIELDS
  tmp$ = STR$(gOpenRecord%)
  EDIT FIELD #_gotoFLD, tmp$, (110,15)-(180,30), _framedNoCR, _centerJust
END FN

LOCAL FN WindowBuild (wndID%)
  LONG IF WINDOW (-wndID%) = 0
    SELECT wndID%
      CASE _dbEntryWIND    : FN BuildEntryWindow
      CASE _dbFindWIND     : FN BuildFindWindow
      CASE _aboutWIND      : FN BuildAboutWindow
      CASE _helpWIND       : FN BuildHelpWindow
      CASE _printWIND      : FN BuildPrintWindow
      CASE _gotoWIND       : FN BuildGotoWindow
    END SELECT
  END IF
  IF wndID% <> _aboutWIND THEN WINDOW #wndID%
END FN

' === APPLE MENU FUNCTIONS ===

LOCAL FN ItemAbout
```

```
  item% = FN ALERT (_aboutALRT, 0)
END FN

LOCAL FN ItemHelp
  FN WindowBuild (_helpWIND)
END FN

LOCAL FN DoAppleMenu (itemID%)
  SELECT itemID%
    CASE _iAbout        : FN ItemAbout
    CASE _iHelp         : FN ItemHelp
  END SELECT
END FN

' === FILE MENU FUNCTIONS ===

LOCAL FN ItemNew
  LONG IF WINDOW (_outputWnd) = _dbEntryWIND
    FN DBBlankRecord
    FN EFtoRecordField
    FN DBWriteRecord
    gOpenRecord% = gMaxRecInFile% + 1
    FN DBBlankRecord
    FN DBWriteRecord
    FN RecordFieldToEF
  XELSE
    tmp$ = "Save new employee file as:"
    gFileName$ = FILES$ (_fSave, tmp$, "Untitled", gWDRefNum%)
    LONG IF LEN (gFileName$) > 0
      FN DBNewEmployeeFile                    'create default file header
      FN DBBlankRecord
      gEmployee.dbName$ = "Empty record"
      gEmployee.dbDeptNum% = _programBTN
      FN DBWriteRecord
      FN WindowBuild (_dbEntryWIND)           'build db window
      FN RecordFieldToEF                      'show it in window
    END IF
  END IF
END FN

LOCAL FN ItemOpen
  LONG IF LEN (gFileName$) = 0
    gFileName$ = FILES$ (_fOpen, "SbDb", , gWDRefNum%)'get file from disk
  END IF
  LONG IF LEN (gFileName$) > 0
    FN DBBlankRecord
    gOpenRecord% = 1
    FN WindowBuild (_dbEntryWIND)             'build db window
    FN DBReadRecord                           'read first record in file
    FN RecordFieldToEF                        'show it in window
  END IF
END FN

LOCAL FN ItemClose
  FN WindowClose (WINDOW (_outputWnd))
  FN UpdateMenus
  gFileName$ = ""
END FN

LOCAL FN ItemPageSetup
  DEF PAGE
END FN

LOCAL FN ItemPrint
  FN WindowBuild (_printWIND)
END FN

LOCAL FN ItemQuit
  gQuit = _true
END FN
```

```
LOCAL FN DoFileMenu (itemID%)
  SELECT itemID%
    CASE _iNew         : FN ItemNew
    CASE _iOpen        : FN ItemOpen
    CASE _iClose       : FN ItemClose
    CASE _iPageSetup   : FN ItemPageSetup
    CASE _iPrint       : FN ItemPrint
    CASE _iQuit        : FN ItemQuit
  END SELECT
END FN

' === EDIT MENU FUNCTIONS ===

DEF FN DoEditMenu (itemID%) USING gEditMenuPtr&

' === RECORD MENU FUNCTIONS ===

LOCAL FN ItemFirstRecord
  gOpenRecord% = 1
END FN

LOCAL FN ItemPrevRecord
  DEC (gOpenRecord%)
  IF gOpenRecord% < 1 THEN gOpenRecord% = gMaxRecInFile%
END FN

LOCAL FN ItemNextRecord
  INC (gOpenRecord%)
  IF gOpenRecord% > gMaxRecInFile% THEN gOpenRecord% = 1
END FN

LOCAL FN ItemLastRecord
  gOpenRecord% = gMaxRecInFile%
END FN

LOCAL FN ItemFindRecord
  FN WindowBuild (_dbFindWIND)
END FN

LOCAL FN ItemGotoRecord
  FN WindowBuild (_gotoWIND)
END FN

LOCAL FN ItemClearRecord
  FN DBBlankRecord
  FN DBWriteRecord
  FOR fieldID% = _dbNameFLD TO _dbPhotoFLD
    EDIT$ (fieldID%) = ""
  NEXT fieldID%
  EDIT FIELD #_dbNameFLD
END FN

LOCAL FN DoRecordMenu (itemID%)
  FN EFtoRecordField                          'save this records data
  FN DBWriteRecord
  SELECT itemID%
    CASE _iFirstRec      : FN ItemFirstRecord
    CASE _iPrevRec       : FN ItemPrevRecord
    CASE _iNextRec       : FN ItemNextRecord
    CASE _iLastRec       : FN ItemLastRecord
    CASE _iFindRec       : FN ItemFindRecord
    CASE _iGotoRec       : FN ItemGotoRecord
    CASE _iClearRec      : FN ItemClearRecord
  END SELECT
  FN DBReadRecord                             'read in new records data
  FN RecordFieldToEF
END FN

LOCAL FN HandleMenuEvent
```

```
  menuID% = MENU (_menuID)
  itemID% = MENU (_itemID)
  SELECT menuID%
    CASE _appleResMenu  : FN DoAppleMenu (itemID%)
    CASE _mFile         : FN DoFileMenu (itemID%)
    CASE _mEdit         : FN DoEditMenu (itemID%)
    CASE _mRecord       : FN DoRecordMenu (itemID%)
  END SELECT
  MENU
END FN

' === DIALOG HANDLERS ===

LOCAL
DIM rect;8
LOCAL FN DrawFrame (showFrame%)
  CALL SETRECT (rect, 210,113,350,275)
  LONG IF showFrame%
    PEN 2,2,,,0
  XELSE
    PEN 2,2,,,19
  END IF
  CALL FRAMERECT (rect)
  PEN 1,1,,,0
END FN

LOCAL
DIM rect;8
LOCAL FN DialogEntryWindow (dlgEvnt%, dlgID%)
  LONG IF dlgEvnt% = _efReturn
    FN ChangeReturnToBtn (dlgEvnt%, _newRecBTN)
  END IF
  SELECT dlgEvnt%
      ' ••• WINDOW EVENTS
    CASE _wndClose
      FN ItemClose
    CASE _wndActivate
      FN UpdateMenus
    CASE _wndClick
      WINDOW #_dbEntryWIND
    CASE _wndRefresh
      'FN ReadEmployeeGraphic
      PEN ,,,,3                                  'set pen to gray pattern
      CALL SETRECT (rect, 10, 10, 360, 280)      'assign rect boundaries
      DEF TITLERECT ("", 0, rect)                'draw patterned rect
      CALL SETRECT (rect, 370, 10, 490, 240)     'repeat for buttons
      DEF TITLERECT ("", 0, rect)
      PEN ,,,,0                                  'reset pen to normal
      LONG IF WINDOW(_efNum) = 0
        FN DrawFrame (_true)
      XELSE
        FN DrawFrame (_false)
      END IF
      ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _newRecBTN
          FN DoFileMenu (_iNew)
        CASE _firstRecBTN
          FN DoRecordMenu (_iFirstRec)
        CASE _prevRecBTN
          FN DoRecordMenu (_iPrevRec)
        CASE _nextRecBTN
          FN DoRecordMenu (_iNextRec)
        CASE _lastRecBTN
          FN DoRecordMenu (_iLastRec)
        CASE _findRecBTN
          FN WindowBuild (_dbFindWIND)
        CASE _clearRecBTN
          FN ItemClearRecord
```

```
        CASE ELSE
          FN RadioBtnHandler% (_programBTN, _officeBTN, dlgID%)
          gEmployee.dbDeptNum% = dlgID%
      END SELECT
      ' ••• FIELD EVENTS
    CASE _efClick
      FN EFClickEvent (dlgID%)
      LONG IF dlgID% = _dbPhotoFLD
        EDIT FIELD #_nil
        FN DrawFrame (_true)
      XELSE
        FN DrawFrame (_false)
      END IF
    CASE _efTab, _efDownArrow, _efRightArrow
      FN TabShiftTabEvents (_efTab, _dbNameFLD, _dbFaxFLD)
    CASE _efShiftTab, _efUpArrow, _efLeftArrow
      FN TabShiftTabEvents (_efShiftTab, _dbNameFLD, _dbFaxFLD)
      ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN

LOCAL FN DialogFindWindow (dlgEvnt%, dlgID%)
  LONG IF dlgEvnt% = _efReturn
    FN ChangeReturnToBtn (dlgEvnt%, _findBTN)
  END IF
  SELECT dlgEvnt%
      ' ••• WINDOW EVENTS
    CASE _wndClose
      FN WindowClose (_dbFindWIND)
    CASE _wndActivate
      FN UpdateMenus
    CASE _wndClick
      WINDOW #_dbFindWIND
    CASE _wndRefresh
      ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _ignoreCaseBTN
          gCaseFlag% = FN CheckBoxHandler% (dlgID%)
        CASE _findBTN
          FN WindowClose (_dbFindWIND)
          FN DBFindRecord
        CASE _cancelBTN
          FN WindowClose (_dbFindWIND)
          gSearch$ = ""
      END SELECT
      ' ••• FIELD EVENTS
    CASE _efClick
      FN EFClickEvent (dlgID%)
      ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN

LOCAL FN DialogHelpWindow (dlgEvnt%, dlgID%)
  SELECT dlgEvnt%
    ' ••• WINDOW EVENTS
    CASE _wndClose
      FN WindowClose (_helpWIND)
    CASE _wndActivate
      FN UpdateMenus
    CASE _wndClick
      WINDOW #_helpWIND
    CASE _wndRefresh
      CLS
```

```
      wndX = WINDOW (_width) : wndY = WINDOW (_height)
      EDIT FIELD #_helpFLD, , (4,34)-(wndX-4,wndY-4)'adjust edit field size
      PLOT 0, 30 TO wndX, 30
      ' ••• BUTTON EVENTS
    CASE _btnClick
      LONG IF dlgID% > _helpSCROLL
        SELECT dlgID%
          CASE _prevHelpBTN
            DEC (gHelpID%)
            IF gHelpID% < _minHelpID THEN gHelpID% = _maxHelpID
          CASE _nextHelpBTN
            INC (gHelpID%)
            IF gHelpID% > _maxHelpID THEN gHelpID% = _minHelpID
        END SELECT
        SCROLL BUTTON #_helpSCROLL, 1
        EDIT$ (_helpFLD) = %gHelpID%
      END IF
      ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN

LOCAL FN DialogPrintWindow (dlgEvnt%, dlgID%)
  LONG IF dlgEvnt% = _efReturn
    FN ChangeReturnToBtn (dlgEvnt%, _printBTN)
  END IF
  SELECT dlgEvnt%
      ' ••• WINDOW EVENTS
    CASE _wndClose
      FN WindowClose (_printWIND)
    CASE _wndActivate
      FN UpdateMenus
    CASE _wndClick
      WINDOW #_printWIND
    CASE _wndRefresh
      ' ••• BUTTON EVENTS
    CASE _btnClick
      SELECT dlgID%
        CASE _thisRecBTN,_allRecBTN,_selectRecBTN
          gPrintFlag = FN RadioBtnHandler% (_thisRecBTN, _selectRecBTN, dlgID%)
        CASE _printBTN
          FN WindowClose (_printWIND)
          FN DoPrinting (@FN DBReadRecord)
        CASE _cancelBTN
          FN WindowClose (_printWIND)
          gPrFirstRec% = _nil
          gPrLastRec% = _nil
      END SELECT
       ' ••• FIELD EVENTS
    CASE _efClick
      FN EFClickEvent (dlgID%)
      gPrintFlag = FN RadioBtnHandler% (_thisRecBTN, _selectRecBTN, _selectRecBTN)
    CASE _efTab, _efDownArrow, _efRightArrow
      FN TabShiftTabEvents (_efTab, _firstPrFLD, _lastPrFLD)
    CASE _efShiftTab, _efUpArrow, _efLeftArrow
      FN TabShiftTabEvents (_efShiftTab, _firstPrFLD, _lastPrFLD)
     ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)
    CASE ELSE
  END SELECT
END FN

LOCAL FN DialogGotoWindow (dlgEvnt%, dlgID%)
  LONG IF dlgEvnt% = _efReturn
    FN ChangeReturnToBtn (dlgEvnt%, _gotoBTN)
  END IF
  SELECT dlgEvnt%
```

```
      ' ••• WINDOW EVENTS
    CASE _wndClose
      FN WindowClose (_gotoWIND)
    CASE _wndActivate
      FN UpdateMenus
    CASE _wndClick
      WINDOW #_gotoWIND
    CASE _wndRefresh
      ' ••• BUTTON EVENTS
    CASE _btnClick
      FN EFtoRecordField                    'save current record
      FN DBWriteRecord
      originalRecNum% = gOpenRecord%        'save current record number
      FN WindowClose (_gotoWIND)            'get new record number
      LONG IF dlgID% = _gotoBTN             'did we want to goto?
        FN DBReadRecord
        FN RecordFieldToEF                  'get new record
      XELSE
        gOpenRecord% = originalRecNum%      ' reset record number
      END IF
      ' ••• FIELD EVENTS
    CASE _efClick
      FN EFClickEvent (dlgID%)
      ' ••• CURSOR EVENTS
    CASE _cursOver, _cursEvent
      FN CursorHandler (dlgEvnt%, dlgID%)

    CASE ELSE
  END SELECT
END FN

LOCAL FN HandleDialogEvent
  dlgEvnt% = DIALOG (0)
  dlgID%   = DIALOG (dlgEvnt%)
  SELECT WINDOW (_outputWClass)
    CASE _dbEntryWIND   : FN DialogEntryWindow (dlgEvnt%, dlgID%)
    CASE _dbFindWIND    : FN DialogFindWindow (dlgEvnt%, dlgID%)
    CASE _aboutWIND     : FN DialogAboutWindow (dlgEvnt%, dlgID%)
    CASE _helpWIND      : FN DialogHelpWindow (dlgEvnt%, dlgID%)
    CASE _printWIND     : FN DialogPrintWindow (dlgEvnt%, dlgID%)
    CASE _gotoWIND      : FN DialogGotoWindow (dlgEvnt%, dlgID%)
  END SELECT
  FN UpdateMenus
END FN
```

--- End of SimpleBase.Incl ---

DialogEvent.Incl

```
' --- HEADER -------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels

' --- CONSTANTS ----------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS

' --- FUNCTIONS ----------------------------------
' === FIELD FUNCTIONS ===

LOCAL FN pEFClickEvent (fieldID%)
  EDIT FIELD #fieldID%
  CURSOR _iBeamCursor
END FN

LOCAL FN pTabShiftTabEvents (dlgEvnt%, startFld%, lastFld%)
  LONG IF dlgEvnt% = _efTab
    fieldID% = WINDOW (_efNum) + 1
```

```
    IF fieldID% > lastFld% THEN fieldID% = startFld%
  XELSE
    fieldID% = WINDOW (_efNum) - 1
    IF fieldID% < startFld% THEN fieldID% = lastFld%
  END IF
  EDIT FIELD #fieldID%
END FN

' === MISC FUNCTIONS ===
LOCAL FN pCursorHandler (cursEvtID%, dlgID%)
  SELECT cursEvtID%
    CASE _cursOver
      SELECT dlgID%
        CASE < 0
          LONG IF (ABS (dlgID%) = WINDOW (_efNum)) AND (WINDOW (_efClass) > 0)
            CURSOR _iBeamCursor
          XELSE
            CURSOR _arrowCursor
          END IF
        CASE > 0
          CURSOR _arrowCursor
        CASE ELSE
          CURSOR _arrowCursor
      END SELECT
    CASE _cursEvent
      CURSOR _arrowCursor
  END SELECT
END FN

' === MISC BUTTON FUNCTIONS ===
LOCAL FN pCheckBoxHandler% (btnID%)
  LONG IF BUTTON (btnID%) = _markedBtn
    BUTTON btnID%, _activeBtn
  XELSE
    BUTTON btnID%, _markedBtn
  END IF
  btnState% = BUTTON (btnID%)
END FN = btnState%

LOCAL FN pRadioBtnHandler% (lowBtnID%, highBtnID%, setBtnID%)
  FOR thisBtn% = lowBtnID% TO highBtnID%
    BUTTON thisBtn%, _activeBtn
    LONG IF thisBtn% = setBtnID%
      BUTTON thisBtn%, _markedBtn
    END IF
  NEXT thisBtn%
END FN = setBtnID%

LOCAL FN pHiliteSelectedButton (btnID%)
  BUTTON btnID%, _markedBtn                    'briefly hilite the correct button
  DELAY _secTick                               'to show it was selected
  BUTTON btnID%, _activeBtn
END FN

LOCAL FN pChangeReturnToBtn (@evntIDPtr&, btnID%)
  FN pHiliteSelectedButton (btnID%)            'hilite correct button
  evntIDPtr&.none% = _btnClick                 'convert return event to _btnClick
END FN = btnID%

' --- SET PROJECT ADDRESSES -----------------------

gCursorPtr&       = @FN pCursorHandler
gEFClickPtr&      = @FN pEFClickEvent
gTabEventsPtr&    = @FN pTabShiftTabEvents
gCheckBoxPtr&     = @FN pCheckBoxHandler
gRadioBtnPtr&     = @FN pRadioBtnHandler
gHiliteBtnPtr&    = @FN pHiliteSelectedButton
gReturnToBtnPtr&  = @FN pChangeReturnToBtn
```

EditMenu.Incl

```
' --- HEADER -------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels

' --- CONSTANTS ----------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS

' ------------------------------------------------
' EDIT MENU ROUTINES
' ------------------------------------------------

CLEAR LOCAL
LOCAL FN DataHandleToScrap (dataH&, dataType&, zeroClipBoard%)
  LONG IF dataH& <> _nil
    LONG IF zeroClipBoard% <> _nil
      scrapH& = FN ZEROSCRAP
    END IF
    sizeOfH& = FN GETHANDLESIZE (dataH&)
    osErr% = FN HLOCK (dataH&)
    LONG IF osErr% = _noErr
      osErr% = FN PUTSCRAP (sizeOfH&, dataType&, [dataH&])
      osErr% = FN HUNLOCK (dataH&)
    END IF
  END IF
END FN = osErr%

CLEAR LOCAL
LOCAL FN ScrapToDataHandle& (scrapType&)
  scrapH& = FN NEWHANDLE (0)
  LONG IF scrapH& <> _nil
    scrapSize& = FN GETSCRAP (scrapH&, scrapType&, offset&)
    LONG IF scrapSize& <= 0
      DEF DISPOSEH (scrapH&)
    END IF
  END IF
END FN = scrapH&

CLEAR LOCAL
LOCAL FN GetPICTHandle&
  tmp$  = EDIT$ (_dbPhotoFLD)
  pict$ = RIGHT$ (tmp$, LEN (tmp$) - 1)
  SELECT LEFT$ (tmp$, 1)
    CASE "%"
      pictH& = FN GETPICTURE (CVI(pict$))
    CASE "&"
      pictH& = CVI(pict$)
    CASE ELSE
      pictH& = _nil
  END SELECT
END FN = pictH&

CLEAR LOCAL
LOCAL FN EditCopy
  pictH& = FN GetPICTHandle&
  LONG IF pictH& <> _nil
    scrapErr% = FN DataHandleToScrap (pictH&, _"PICT", _true)
    DEF DISPOSEH (pictH&)
  END IF
END FN

CLEAR LOCAL
LOCAL FN EditCut
  FN EditCopy
  EDIT$ (_dbPhotoFLD) = ""
  gEmployee.dbPictID% = _nil
  DEF DISPOSEH (gPictH&)
```

```
END FN

CLEAR LOCAL
LOCAL FN EditPaste
  pictH& = FN ScrapToDataHandle& (_"PICT")
  LONG IF pictH& <> _nil
    DEF DISPOSEH (gPictH&)
    gPictH& = pictH&
    EDIT$ (_dbPhotoFLD) = &gPictH&
    gEmployee.dbPictID% = gOpenRecord%
  END IF
END FN

CLEAR LOCAL
LOCAL FN EditClear
  EDIT$ (_dbPhotoFLD) = ""
  gEmployee.dbPictID% = _nil
  DEF DISPOSEH (gPictH&)
END FN

CLEAR LOCAL
LOCAL FN EditSelectAll
  LONG IF WINDOW (_efClass) > 0                    'are we in an edit field?
    SETSELECT 0, _maxInt
  END IF
END FN

CLEAR LOCAL
DIM 255 tmp$
LOCAL FN EditExportRecord
  FOR count% = _dbNameFLD TO _dbFaxFLD
    calcHSize% = calcHSize% + LEN(EDIT$(count%)) + 1
  NEXT count%
  offset% = 0
  recordH& = FN NEWHANDLE (calcHSize%)
  LONG IF (recordH& <> 0) AND (SYSERROR = _noErr)
    osErr% = FN HLOCK (recordH&)
    LONG IF osErr% = _noErr
      FOR count% = _dbNameFLD TO _dbFaxFLD
        tmp$ = EDIT$(count%)
        LONG IF count% < _dbFaxFLD
          char$ = CHR$(_tab)
        XELSE
          char$ = CHR$(_cr)
        END IF
        tmp$ = tmp$ + char$
        size% = LEN (tmp$)
        BLOCKMOVE @tmp$+1, [recordH&] + offset%, size%
        offset% = offset% + size%
      NEXT count%
    END IF
    osErr% = FN HUNLOCK (recordH&)
    osErr% = FN DataHandleToScrap (recordH&, _"TEXT", _true)
    DEF DISPOSEH (recordH&)
  END IF
END FN

CLEAR LOCAL
DIM   tmp$
DIM 3 char$
LOCAL FN EditImportRecord
  scrapH& = FN ScrapToDataHandle& (_"TEXT")
  LONG IF scrapH& <> _nil
    strPtr& = @tmp$
    charPtr& = @char$ + 1
    char$ = CHR$(_tab)
    startPos& = _nil
    offset& = FN MUNGER (scrapH&, startPos&, charPtr&, 1, _nil, _nil)
    LONG IF offset& > _nil
      fieldID% = _dbNameFLD
```

```
      DO
        size% = offset& - startPos&
        POKE strPtr&, size%
        BLOCKMOVE [scrapH&] + startPos&, strPtr&+1, size%
        EDIT$ (fieldID%) = tmp$
        INC (fieldID%)
        startPos& = offset& + 1
        offset& = FN MUNGER (scrapH&, startPos&, charPtr&, 1, _nil, _nil)
      UNTIL (offset& < 0) OR (fieldID% = _dbFaxFld)
      size% = FN GETHANDLESIZE (scrapH&) - startPos&
      POKE strPtr&, size%
      BLOCKMOVE [scrapH&] + startPos&, strPtr&+1, size%
      EDIT$ (fieldID%) = tmp$
    XELSE
      item% = FN NOTEALERT (_notRecordALRT, 0)
    END IF
    DEF DISPOSEH (scrapH&)
  END IF
END FN

LOCAL FN pDoEditMenu (itemID%)
  SELECT itemID%
    CASE _iCut         : FN EditCut
    CASE _iCopy        : FN EditCopy
    CASE _iPaste       : FN EditPaste
    CASE _iClear       : FN EditClear
    CASE _iSelectAll   : FN EditSelectAll
    CASE _iCopyRec     : FN EditExportRecord
    CASE _iPasteRec    : FN EditImportRecord
  END SELECT
END FN

' ••• get global function pointers

gEditMenuPtr& = @FN pDoEditMenu
```

Printing.Incl

```
' --- HEADER -------------------------------------
INCLUDE FILE _aplIncl
COMPILE 0, _strResource_macsBugLabels

' --- CONSTANTS ----------------------------------
GLOBALS "SimpleBase.glbl"
END GLOBALS

' --- FORWARD FUNCTIONS --------------------------

LOCAL FN DBReadRecordTemplate
END FN

' --- FUNCTIONS ----------------------------------
LOCAL
DIM rect.8
LOCAL FN PrintRecord (pgVOffset%)
  xOffSet% = 150
  vOffset% = 15
  TEXT _geneva, 9, 1
  ' ••• PRINT FIELD TITLES
  FOR count% = _dbNameFLD TO _dbFaxFLD
    tmp$ = UCASE$ (STR# (_dbEntryWIND_fieldSTR, count%))
    PRINT%(xOffSet% + _gutterAdj, pgVOffset% + vOffset%) tmp$
    vOffset% = vOffset% + 15
  NEXT count%
  ' ••• PRINT FIELD DATA
  TEXT _geneva, 12, 0
  PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 15) gEmployee.dbName$
```

```
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 30) gEmployee.dbAddr$
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 45) gEmployee.dbCity$
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 60) gEmployee.dbMyState$
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 75) gEmployee.dbZip$
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% + 90) gEmployee.dbPhone$
    PRINT%(xOffSet% + _gutterAdj + 80, pgVOffset% +105) gEmployee.dbFax$
    ' ••• PRINT PICTURE & SEPERATOR
    CALL SETRECT (rect, _gutterAdj,pgVOffset%, 130 + _gutterAdj, pgVOffset%+152)
    PICTURE FIELD #100, %gEmployee.dbPictID%, @rect, _statFramed, _cropPict
    PEN ,,,,3
    PLOT 0, pgVOffset% + 165 TO 600, pgVOffset% + 165
    PEN ,,,,0
  END FN

  LOCAL FN PrintManyRecords (firstRec%, lastRec%, readRecPtr&)
    pgVOffset% = 10
    pageNum% = 1
    recCount% = 0
    DO
      gOpenRecord% = firstRec%
      FN DBReadRecordTemplate USING readRecPtr&;
      FN PrintRecord (pgVOffset%)
      INC (firstRec%)
      INC (recCount%)
      LONG IF (recCount% MOD 4) = 0
        PRINT%(_gutterAdj, pgVOffset% + 180) "PAGE#";pageNum%
        INC (pageNum%)
        pgVOffset% = 10
        IF recCount% < lastRec% THEN CLEAR LPRINT
      XELSE
        pgVOffset% = pgVOffset% + 180
      END IF
    UNTIL firstRec% > lastRec%
  END FN

  LOCAL FN pDoPrinting (readRecPtr&)
    DEF LPRINT
    LONG IF PRCANCEL = 0
      'TRON p
      oldRecNum% = gOpenRecord%
      resRef% = USR OPENRFPERM (gFileName$, gWDRefNum%, _fsCurPerm)
      CURSOR _watchCursor
      ROUTE _toPrinter
      SELECT gPrintFlag
        CASE _thisRecBTN
          FN PrintRecord (10)
          ROUTE _toScreen
          CLEAR LPRINT
        CASE _allRecBTN
          FN PrintManyRecords (1, gMaxRecInFile%, readRecPtr&)
        CASE _selectRecBTN
          FN PrintManyRecords (gPrFirstRec%, gPrLastRec%, readRecPtr&)
      END SELECT
      ROUTE _toScreen
      CLOSE LPRINT
      IF resRef% THEN CALL CLOSERESFILE (resRef%)
      gOpenRecord% = oldRecNum%
      FN DBReadRecordTemplate USING readRecPtr&;
      CURSOR _arrowCursor
    END IF
  END FN

  ' --- SET PRINT ADDRESS -------------------------
  gDoPrintPtr&    = @ FN pDoPrinting
```

SimpleBase.Main

```
' --- HEADER -------------------------------------
RESOURCES "SimpleBase.rsrc", "APPLFbSb"
COMPILE 0, _strResource_macsBugLabels
OUTPUT FILE "SimpleBase apl"

' --- CONSTANTS ----------------------------------

GLOBALS "SimpleBase.glbl"
END GLOBALS

' --- INCLUDES -----------------------------------

INCLUDE "DialogEvent.Incl"
INCLUDE "EditMenu.Incl"
SEGMENT
INCLUDE "SimpleBase.Incl"
INCLUDE "Printing.Incl"

' --- FUNCTIONS ----------------------------------

LOCAL FN CheckIncomingFiles
  maxFiles% = 1
  doWhat% = FINDERINFO (maxFiles%, gFileName$, fileType&, gWDRefNum%)
  LONG IF (maxFiles% > 0) AND (fileType& = _"SbDb")
    SELECT doWhat%
      CASE _openFiles
        FN ItemOpen
      CASE _printFiles
        FN ItemOpen
        FN ItemPrint
    END SELECT
  END IF
END FN

LOCAL FN Initialize
  EDIT = 2
  WINDOW OFF
  MINWINDOW 240, 120
  MAXWINDOW SYSTEM(_scrnWidth)-20, SYSTEM (_scrnHeight) - 50
  gQuit = _false
  gHelpID% = _minHelpID
  LONG IF SYSTEM (_machType) < _envMacPlus
    item% = FN NOTEALERT (_machErrALRT, 0)
    END
  XELSE
    LONG IF SYSTEM (_sysVers) < 605
      item% = FN NOTEALERT (_sysErrALRT, 0)
      END
    END IF
  END IF
  FN CheckIncomingFiles
END FN

' --- MAIN LOOP ----------------------------------

FN Initialize
ON MENU FN HandleMenuEvent
ON DIALOG FN HandleDialogEvent

DO
  HANDLEEVENTS
UNTIL gQuit
END
```

Index

S

T

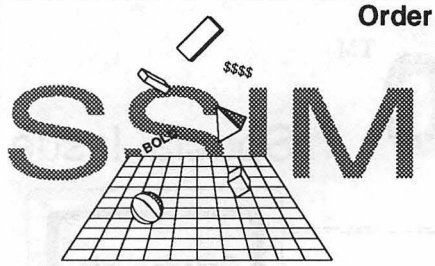
U

V

W

Z

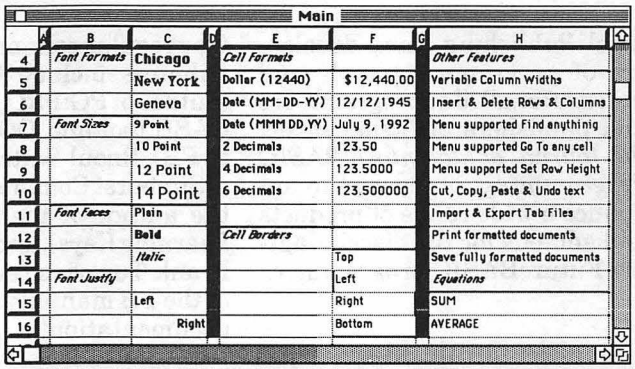
FutureBASIC Source Code



Order SSIM now and get the following: **$69.95**

- Cell Interface Module (CIM) a quick way to put together a scrolling array of cells.
  - Everything is linked to a single rectangle.
  - Make grid any size, put it in any window.
  - Control column widths
  - Control column formats such as color, font, face, etc.
  - All the information is stored in a simple text array.
  - Get source code and the compiled application.

- Spread Sheet Interface Module (SSIM) - A full blown spread sheet engine.
  - Control font, size, color, border, justification, and format for each cell.
  - Variable column widths
  - Insert and delete columns and rows
  - Import and Export ASCII tab files
  - Save fully formatted files
  - Formula and equation support
  - 30,000 rows by 30,000 columns
  - Text Cut, Copy, Paste, and UNDO!
  - Supports B&W and Color
  - Print fully formatted files
  - Copy and paste ranges of cells



FN SOLVE $40

Features:

- A single include file and 6 global variables
- Returns formatted values
- Handles ...
  - ...up to 50 pairs of **brackets** (expandable by changing globals)
  - ...**functions**, comes with TAN, SIN, etc. (you can add your own)
  - ...+, -, *, / and ^
  - ... **variables** defined in a global array
  - ...**scientific notation**

How it works:

1) a$="2.3 * SIN(3E4-6.33)/34.6^-23.4"
2) Answer$=FN Solve$(a$)

(you can include variable names in a$ if you define them before hand in a global array.)

Solves this equation:

2.3 * (CSC(3E4-6.23) / (TAN(3.2+5))^1.4)

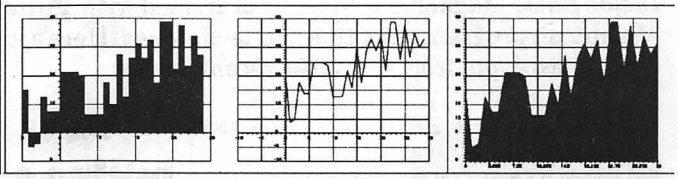
FN GRAPH $30

Features:

- Everything is tied to one rectangle, scale or move your plot by changing a rectangle coordinate.
- Supports five types of graphs: Scatter, Line, Bar, Thin bar and Area.
- Control scale, number of tick marks, plot grid, plot box, point markers, color, axis crossing points.
- Supports COPY graph

How it works:

1) Move your data into the plotting array
2) Set your graph type (gChart=_lineGraph)
3) Set your rectangle (CALL SETRECT(r,0,0,50,50))
4) Let the code set the plotting parameters (FN MakeChart)
5) Draw the graph (FN Graph)



With all **...by Design** software you get:
Easy to follow manual.
1 month of FREE technical support
Source code and compiled applications
No runtime fee

To Order

Call:
(608) 831-5259

Write:
...by Design, Inc.
5700 Highland Way, Ste.#201
Middleton, WI 53562

VISA/MasterCard

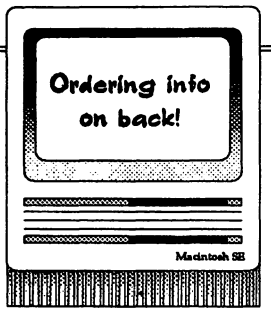
Shipping on all orders:
$3 US, $15 Foreign

Get the only publication that speaks FutureBASIC!

Inside Basic™

The Journal of Macintosh BASIC Programming

Special Issue



We speak FutureBASIC!

Ariel Publishing, Inc., developers of programming tools and utilities for all Mac programmers and publishers of *Inside Basic, the Journal of Macintosh BASIC Programming,* is pleased to announce a wide range of products and add-ons for the discriminating FutureBASIC programmer.

Inside Basic Magazine

Our flagship product for BASIC developers, IB is the primary source of FutureBASIC information available anywhere. Not only does our publication cover FB in depth, but it also **teaches Mac programming.** You'll learn how to use FB more intelligently, too (of course!), but you'll also get many, many tips and techniques that will help you to take best advantage of the Mac toolbox.

Our regular columnists and contributors include Chris Stasny (author of PG:PRO, the awesome CASE tool and OOPs extension to FB), Raoul Watson (presently of Sunburst Communications and the author of the Mac Muppet Learning Keys), and Zedcor's own Frank Turovich (author of most of the FB manuals and technical documentation!).

IB is a monthly journal and costs $49.95 for one year, $94.95 for two years.

Inside Basic on Disk, a companion disk subscription, contains all of the article text and source code from the magazine (as well as the latest and greatest utilities for FB programmers). *IB on Disk* is $49.95 for one year, $89.95 for two years.

CDEF-City™

This package of five control definitions not only adds pizazz t your programs, but also save *tons* of time when you want t perform several tricky tasks lik creating animated buttons, put ting a menubar inside a windo or turning part of a PICT into button.

And you need not worry that thes controls are a pain to work with either. FB programmers have th luxury of being able to build them with FB's own BUTTON state ment (and handle them with FB's standard DIALOG functions) CDEF-City is $49.95 ($39.95 fo IB subscribers).

Tips, Tricks, & Techniques Galore!

Inside Basic Magazine is literally crammed with FutureBASIC and Macintosh programming tips and techniques. Here's just a couple of quickie excerpts from our most recent issues:

March 1992
Installing a ? menu

```
' Note: Gestalt Mgr only available in
' System 6.04 or later.
HelpMgr& = FN GESTALT(_"help")
LONG IF HelpMgr&
 OSErr=FN HMGETHELPMENUHANDLE(mhndl&)
  LONG IF OSErr = 0 AND mHndl& <> 0
   CALL APPENDMENU(mHndl&,"My Very Own Help Menu for System 7")
  END IF ' install help elsewhere for System 6
END IF
```

continued on p.2

QDFx™

If you envy HyperCard's ability to do snazzy wipes, dissolves, and transitions from one image to another, then you need QDFx™. This package includes 16 different visual effects, including Venetian Blinds, Iris In, Slide, Zoom Box, Dissolve, Bubbles, Horizon, Cascade, and many more.

QDFx allows you to bring a new

continued on p.2

ips & Tricks, cont.

April 1992
A MultiColumn Scrolling List

```
ORDINATE WINDOW : CLS
t,l,b,r
lWdth = 45 : rowHt = 12:TEXT 1,9,0,0

NG FN drawCell(row,col,theTxt$,seltd)
. = (row-1)*rowHt : b = t + rowHt
l = (col-1) * colWdth : r = l + colWdth
ALL TEXTBOX((@theTxt$)+1,LEN(theTxt$),t,0)
F seltd THEN CALL INVERTRECT(t)
ALL INSETRECT(t,-1,-1)   ' create frame
ALL FRAMERECT(t)          ' and draw it
D FN

emo"

electedRow = RND(10)
OSUB "Show List"
TIL LEN(INKEY$) OR FN BUTTON  ' mouse click or key to end
D
how List"
R row = 1 TO 10
OR col = 1 TO 5
 IF row = selectedRow THEN seltd = _ZTrue ELSE seltd = _False
 theTxt$ = "("+STR$(row)+","+STR$(col)+")"
FN drawCell(row,col,theTxt$,seltd)
NEXT
XT
TURN
```

Ariel Goodies, cont.

image into all or part of your window in a very artistic and pleasing fashion. It is also extraordinarily easy to use from FutureBASIC (just CALL "QDFx" with the proper parameters!). QDFx retails for $49.95, but *Inside Basic* subscribers can get it for $39.95.

Odd I/O™

A "snd " resource library, Odd I/O contains over 3.25 megabytes of royalty free sound data. It includes the beautiful (Mozart, etc.), the funny and bizarre ("totallymondognarly"), and the practical (the alphabet and the numbers 0-9). Odd I/O retails for $19.95 ($14.95 with an IB sub). Since FutureBASIC plays "snd " resources asynchronously with its built in SOUND statement, these sounds are easy to use in your own programs.

To Order:

(509) 923-2249 (voice AND fax)
or clip the form below and write:
Ariel Publishing, Inc., P.O. Box 398, Pateros, WA 98846

Name _____
Address Line 1 _____
Address Line 2 _____
City, State, Zip _____

NOTE: Non-North American customers please add $48 for 1st Class Postage or $18 for 3rd class postage per year, per magazine subscription.

IB on Disk is only shipped 1st Class, but the cost is $18 per year, per disk subscription.

Inside BASIC Magazine ❑ 1 year: $49.95 ❑ 2 years: $94.95
Inside BASIC on Disk ❑ 1 year: $49.95 ❑ 2 years: $89.95

❑ CDEF-City..$49.95 ($39.95) ❑ QDFx.$49.95($39.95) ❑ Odd I/O..$19.95 ($14.95)
prices in parenthesis are the discounted price in conjunction with a subscription

Total of all products ordered: ..... $_____

Shipping and handling:	**Software**	**IB Magazine**	**IB on Disk**
All North American destinations	free	free	free
Non-North American	$5.00	$18.00 per year (3rd class) $48.00 per year (1st class)	$18.00 per year (1st class)

Total Shipping and Handling ..... $_____
Sales Tax (Washington state residents only add 7.5%)..... $_____

Grand Total ..... $_____

Method of payment: ❑ Visa ❑ MC ❑ Check ❑ Purchase Order ❑ Bill Me (net 90)

Card or PO#_____ Exp. Date_____ Signature_____

Need Random Files? **Need Fast Searches?**

consider

B-Tree HELPER™

from

(M)agreeable software

B-Tree HELPER

Gets space in a file in fixed length blocks. You determine block size at file creation, from 9 to 16,388 bytes.

Releases space back to the free block pool.

Expands the file as necessary, up to the maximum available on the media.

Contracts files when possible.

B-Tree HELPER

Inserts keys in one or more B-Trees in one or more files. Keys may be strings, integers, or any other data.

Finds the keys equal to, less than, or greater than a given value in a few hundredths of a second.

Finds lists of records whose keys are equal to, less than, or greater than a given value or in a range of values.

Deletes keys.

B-Tree HELPER Saves You Time

It could take you two or more weeks to write a file space management system.

It would take you four or more weeks to write a file-based B-Tree management system.

B-Tree HELPER is ready to run.

B-Tree HELPER Saves You Money

For **only $75.00** you can save weeks of typing, testing, retyping, retesting...

B-Tree HELPER is a set of code resources you call from your FutureBASIC programs.

You pay **no** license fees or royalties.

What Do Our Customers Say?

"Both your product and way of doing business are worthy of a kind word. ...Thanks for a nice program at a nice price."
– Ed Ringel, Waterville ME

"Tu sei troppo simpatico. Tante grazie per la pazienza e buon lavaro."
– Antonio Cocco, Caserta, Italy

"B-Tree HELPER works great! Multiple keys and data can be stored in one Mac file; and data can be anything: pictures, edit fields, etc. It's one of the best database products I've seen – especially for the price."
– John W. Roberts, Palo Alto CA

"B-Tree HELPER is a great product, and is an exceptional value for the money, especially when you consider its capabilities and the development time that using it has saved."
– John Sidney-Woollett, GTec Systems, London, England

Order B-Tree HELPER™ Today

___ Please send me FutureBASIC code resources for B-Tree HELPER 2.1 $75.00

___ Please send me the THINK C source code for B-Tree HELPER 2.1 ($150.00 separately) $100.00

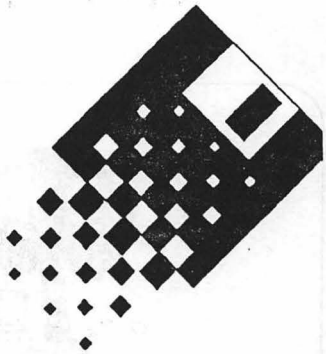
____ Please send me the Pascal source code for B-Tree HELPER 2.1 ($150.00 separately) $100.00

Enclose check or money order in U.S. Dollars. For other currencies, please write, call, or e-mail: MAGREEABLE (GEnie, AppleLink, or America OnLine), or 72167,1700 (CompuServe).

Name _____
Company _____
Address _____
City _____ State ____ Zip _____
Country _____

Send to: (M)agreeable software, Inc., 5925 Magnolia Lane, Plymouth MN 55442-1573, Ph. (612) 559-1108.

PG:PRO



Not everyone is a professional
ɔgrammer. But each of us has
acquired knowledge in specific
fields through study or job
experience or just because we
ıappened to be standing under
e right tree when an apple fell.

f you have discovered a unique
way to store, analyze or
calculate information (or
perhaps you have an idea for a
ɔrld class game) there is only
one (good) way to share it: a
Macintosh application. And
until now, there was only one
way to create it: spend years
learning to program.

PG:PRO has changed all of
that. Because now you don't
have to be a professional
programmer to create a
professional program. Just let
the **PRO** handle interface
operations and spend your time
ɔrking on the things that made
ɔu a professional to begin with.

MACWORLD MW★★★★

Staz Software has produced the equivalent of an object-oriented programming (OOP) environment that doesn't require previous OOP experience. This is an amazing accomplishment in its own right, but the speed and compactness of the code are phenomenal.

- MacWorld ► July 1993

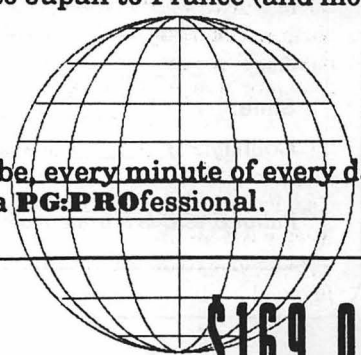
PG:PRO makes a great partner for FutureBASIC and makes it easy to design applications quickly without getting mired in the details of managing the interface.

- Byte ► June 1993

PG:PRO provides the means for even the most non-professional programmers to create a solid Mac interface in minutes.

- Inside BASIC ► June 1992

Lots of folks are using the **PRO**. There are hobbiests, scientists, engineers, artists, gamers The **PRO** has even cut new paths into the HyperCard dominated markets of higher education. We cover the U. S. from the University of Florida to Washington state's Battelle Labs. And we span the globe from Australia to Japan to France (and most points in between.)



Somewhere on the globe, every minute of every day, the sun is shining on a **PG:PRO**fessional.

STAZ SOFTWARE

$169.00

11A LEISURE TIME DRIVE • DIAMONDHEAD, MS 39525 • 601•255•7085 • FAX 601•255•7086

ORDER FORM

***Sentient Fruit*™**

MACINTOSH CONSULTING • PROGRAMMING • DOCUMENTATION

PO BOX 13362 • TUCSON • AZ 85732-3362

The first programming book for...

Learning FutureBASIC

Macintosh BASIC Power

by L. Frank Turovich

Designed for people already familiar with BASIC, but who want to program professional-quality Macintosh applications. Inside you'll learn...

• Techniques for faster, bug-free programming • How to create and manage up to 63 program windows • How to add and handle buttons, edit and picture fields • Why events are vital and how to program for them • Amazing printing techniques for text and graphics • How to design and use common resources effortlessly (MENU, ALRT, CURS, DLOG, DITL, ICN#, STR#, and many more) • How to handle files, records, and folders like a pro • Hundreds of other useful tips and suggestions • A complete SimpleBase program shows you how!

Wow *– I can't wait, please send my copy of Learning FutureBASIC: Macintosh BASIC Power (ISBN 0-9639552-0-9) now for $39.95 US to...*

Name: _____

Address: _____

_____

City: _____

State: _____ Postal Code: _____

Country: _____

FutureBASIC is a trademark of Zedcor, Inc.

Sales Tax:
Arizona residents please include 7% sales tax.

•

Shipping:
USA $6.00,
Canada $8.00,
Europe $14.00,
Pacific Rim $17.00
Payable in US funds by US check or Intl. money order.

•

Please allow 4-6 weeks for processing and shipping.

Sentient Fruit • P.O. Box 13362 • Tucson, AZ 85732-3362
ISBN 0-9639552-0-9
$39.95 USA