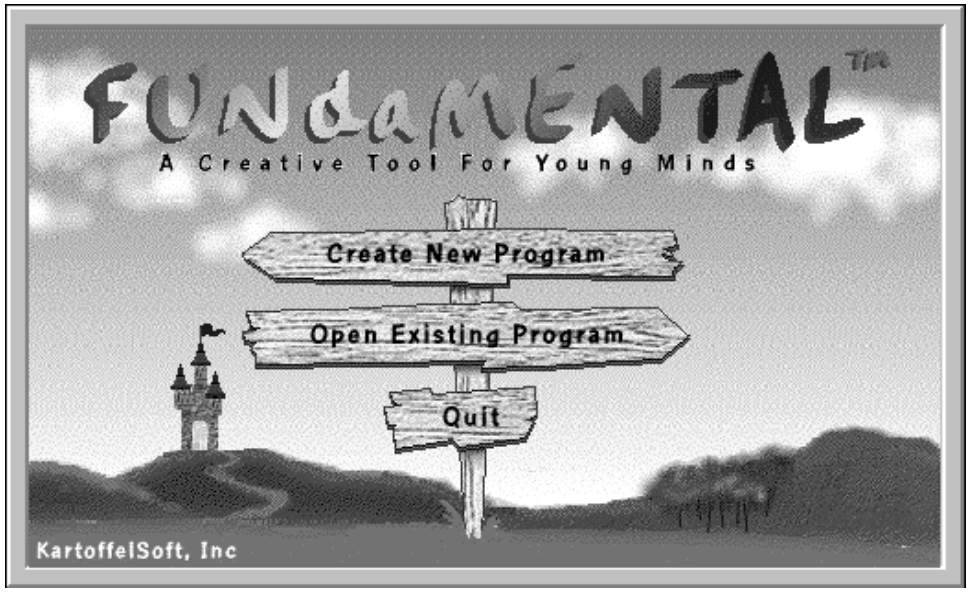
# The FUNdaMENTAL
# Teacher's Manual and Curriculum Guide




## Welcome to
## Our Programming Environment for Kids!

For further information, please contact KartoffelSoft, Incorporated, at:

3475 Edison Way, Suite H
Menlo Park, CA 94025
(415) 306-0670

or contact your licensed KartoffelSoft representative.

# ----- Author's Note -----

When I was first asked to learn FUNdaMENTAL as "a typical teacher" would, I have to admit I was a little nervous. I'd always purposefully avoided anything to do with the inner workings of my computer. I just clicked buttons and pushed keys, and left the rest to the "genie in the box." Actually, I guess I'd always known that a breed of humans (not genies) called programmers were responsible for making the computer do the things it did. I'd just never imagined that I could become a programmer myself. (Suffice it to say, I'm the only one at Kartoffelsoft, Inc. sporting a degree in Latin American Literature on my undergraduate diploma) Imagine my delight when I did, indeed, become a FUNdaMENTAL programmer. I was truly proud.

When I was offered the opportunity to teach FUNdaMENTAL to a group of seventh and eighth graders in the Stanford Tennis and Tutorial program for at-risk students, I jumped at the chance. I couldn't wait to share my enthusiasm for the magic of the programming process with young people. Through their frustrations and triumphs, those bright and intrepid youngsters helped me unravel the general sequence of lessons which form the basis for this book. Highlights of my experiences appear in the Teacher's Journal sections throughout the manual.

When I was asked to write the Teacher's Manual, my jaw dropped practically past my belly button. I tried to maintain my professional composure. But inside I was thinking, "Okay so I *learned* it, and I *taught* it, but now I'm supposed to *write the book!!?* I'm going to need *a lot* of help…"

"We want it to be in your voice–a real teacher's voice, like the one in your journals. Not the typical, dry user's manual…" the young man in front of me was saying.

"I'm going to need *a lot* of help!" I finally squeaked out loud.

I cannot begin to offer enough thanks to all of those who gave the heaps and heaps of help necessary to bring this book into existence. Sarah Morse provided initial inspiration and momentum. Marlo Khan, Sam Birney, and Thai Bui reviewed the initial drafts and shored up the structural and technological underpinnings. After that, Stephen Wu took over as my technical guru, patiently answering my confused, and often repeated, questions about everything from memory to pixels. Shane Reilly worked out an early layout design for the book. A group of valiant beta-testers took on an initial "final" draft. Their input on the practice sections and the overall tone was invaluable. And Joe Davila was there throughout to provide encouragement and vision when I fell prey to what he so aptly called "analysis paralysis." In the end, he took on the final layout effort which resulted in the book you now hold.

And now, after many months and lots and *lots* of help, this book is finally in your hands, which means you're about to begin the adventure of learning and teaching FUNdaMENTAL yourself. It is, indeed, not your typical user's

manual. As well as providing instruction and practice, I hope that this manual offers the companionship and inspiration that you'd get from a close colleague and friend as you set out to bring this exciting new learning tool into the lives of your students. And in the true spirit of collegiality, I hope that this book is the beginning of a dialogue in which you share with us at KSI, Inc. your own stories of challenge and triumph as your students demonstrate the true power of programming.

Read on, and enjoy!

Janet Ellman
jellman@kartoffelsoft.com

# TABLE OF CONTENTS

# ILLUSTRATION GUIDE

## Screen Images

# "What's Going On Inside the Computer?" Diagrams

# FROM THE CREATOR OF FUNdaMENTAL

---

When I was in sixth grade, I had my first real exposure to computer programming. It was then that a classmate and I spent several months tinkering with a program called Logo. To us, it was a nifty game that made a little "turtle" draw colorful lines on the screen if you gave it the right instructions. We became quite proficient at this, and soon had a Godzilla-like creature chasing a Ferrari across the screen. After a few months of tinkering we grew tired of Logo, not knowing that the skills we were mastering were not just useful to turtles, but would be honed and utilized for the rest of our lives.

It was more than seven years later when computer programming and I next crossed paths. This time, it was in an introductory programming class at Stanford University. About a week into the course, we began fiddling with Karel the Robot, a programming language that controls a little robot in his own "microworld." That's when it dawned on me: I had done this before. Karel was nothing more than a boring version of Logo. Only then did I realize that what I had been learning all those years ago was how to *program a computer*.

As the quarter progressed, I was taught how to solve problems "from the top down" (using a technique called *decomposition*), and to think logically and "step-wise" like a computer. I learned to "de-bug" programs with a methodical and swift attack. These were all skills to which I'd been unwittingly introduced way back in the sixth grade, and which I'd been using ever since to write seventh grade English essays, solve algebra problems, unravel logic puzzles, and even pursue my affinity for baking.

If only I had known back than that I had been learning all those things, I would have learned them ten times as well. I would have wanted to learn *how* a computer works, and *how* I could make it do really cool things. It seemed that an early opportunity to focus my creative energies and refine my problem-solving skills had been squandered.

In 1994, when I began searching for a thesis topic, my first thought was to combine two of my main passions: kids and computers. That's when I recalled my revelation of freshman year and my stunted Logo career. What kids should really be using, I thought, is a computer language that *is* a computer language, and not one that is hidden behind a game. How many other kids, like me, had failed to see Logo for its true magnificence because they were too busy playing with a turtle?

I was motivated by my firm belief that children want to learn how things work, and by my conviction that there are actually more effective ways to get a child's attention than video games (like a piece of paper and crayons).

I set out to write my own computer language for children that didn't try to be something else.

The result is FUNdaMENTAL, a computer language for kids that is simply that: a programming language which empowers the child (or anyone else with the time and creativity) to make the computer do whatever s/he wants it to do. We'd like to think that it is proof that "fun" and "mental" can be in the same sentence for a kid.

All of us at KartoffelSoft would like to thank you for taking the time and effort to learn FUNdaMENTAL. Unlike many "educational" products on the shelves today FUNdaMENTAL cannot be learned by simply taking it out of the package and clicking around. This product is designed to be a learning tool, not a productivity tool; the focus should be not only on *what is being created* but also on *the process of creating*. In other words, learning FUNdaMENTAL is *using* FUNdaMENTAL. It involves dozens of mini "mental conquests" on the learner's part—the internalization of new concepts and possibilities—which take hours, not seconds. As you embark upon the process of preparing to bring FUNdaMENTAL into the lives of your students, you'll need to follow all the same advice you'll soon be giving them: practice, experiment, turn mistakes into features, and, most of all, *trust yourself*.

More than anything else FUNdaMENTAL is about conquering a technology that is unknown and therefore intimidating. You'll need to model for your students what it means to be an aggressive learner who refuses to give up. Learning FUNdaMENTAL is difficult, but it is supposed to be. Like all truly worthwhile endeavors, it is both frustrating and exhilarating. It is an experience that will transform the way you and your students approach technology and problem solving.

I sincerely hope you enjoy learning FUNdaMENTAL. More importantly, I hope you take this opportunity to share the magic of programming with the children in your life, those professional learners and creators who never cease to amaze us all.

Justin Shelby Kitch
President & Cofounder
KartoffelSoft, Inc.
1997, Menlo Park, CA
jkitch@kartoffelsoft.com

# INTRODUCTION

## Welcome to Programming with FUNdaMENTAL!

## ----- Teacher's Journal -----

"HEY!" I called out to my husband. "You have to come see this! You won't believe it!"

He put down his magazine and came in right away. After all, it wasn't like me to be hogging the computer on a Saturday afternoon.

"Look!" I said with a goofy Cheshire Cat grin spread all over my face. "Just look at this!"

I clicked a button on the screen and watched as the monitor cleared and revealed an empty, white window. A moment later, a little green frog appeared from the left and hip-hopped a zigzag path across the white field and out of sight past the upper right-hand corner. As soon as he was gone, another one followed.

"Okaaaaaay," my husband said, struggling to maintain a diplomatic little smile.

"Don't you get it?" I said. "I MADE this! I WROTE THIS PROGRAM!! Just think what you could do with this in the classroom."

My husband blinked at the procession of little frogs hopping across the screen. (I think he may have resisted the impulse to feel my forehead. Was this the same woman who usually hovered around outside the study doorway, saying things like, "Why don't you turn that thing off and go outside?")

"That's great, Jan" he finally said, and turned to go back to his reading.

I peeled my eyes off my creation and turned to stare after him as he went down the hall. It dawned on me that despite all the time he'd spent using the computer (for games and grade sheets and word processing and taxes), he had never really thought about what went into making all those programs he used. He just didn't get it.

But the most amazing part of it was, I did! I really got it!

## Who Can Program?

If you're a dedicated educator, eager to help guide your students toward the challenges of the 21st Century…and still a little afraid to turn on that computer in the back of the classroom, the programmer in the teacher's journal above could be you!

I know, because the journal is mine, and, not long ago, I was still a little shaky about computer basics myself (details like "Where's the ON switch?") Yet now I can say this with confidence: I am a computer programmer! The fact that I still sometimes get lost on my way to the Toolbar does not change that amazing truth.

And that's not all. Within a few weeks of starting to learn FUNdaMENTAL myself, I brought it into the classroom and became a programming teacher!

## What Is Programming, Anyway?

I certainly had no idea before I got started using FM. When I first came on as resident teacher at KartoffelSoft, Inc., I just tried to paste a comprehending look on my face and nod professionally whenever one of the programmers started talking about "modularity" or "ub-asks" or "code." I really had no concept of what programming was. Just in case you're in the same boat, let me tell you what I've learned.

In a nutshell, *programming is the art and science of giving instructions to computers in order to achieve human goals*.

(In my own first programming effort, for example, I was trying to make a little animated tribute to Kermit the Frog.)

Every computer program that you buy in the store exists because someone, somewhere sat down and typed in a looooooong list of instructions called commands, in a language that the computer understands. This list is the program. The program is like the recipe for everything you see, hear, and do when you use a piece of software. Collectively, all of the commands that make up a particular program are called the program's code.

## What Is FUNdaMENTAL?

FUNdaMENTAL is both a programming language and an "environment" in which beginners of all ages can write and run their own computer programs.

When you want to write a program, you start out in the Task window where you actually write the programs, using the keyboard to type in instructions for the computer to follow. Then, by clicking the Play button on the Toolbar, you can see what happens when the program runs in the Playground and Conversation windows, as the computer

carries out all your instructions. You can switch back and forth between the writing mode and the running mode as often as you like.

When you're writing a program, you'll notice that you'll click that Play button pretty frequently, as a way to get a sense of how your program is coming along. If you like what you see, then the Playground and Conversation windows become the public forum in which you can "publish" your program by showing it off to other people.

## What Do FM Commands Look Like?

There are many different computer languages, and each one of them uses a different code for "communicating" with the computer. In FUNdaMENTAL, the commands are all made of words, sometimes two and sometimes three. Here are some examples of FUNdaMENTAL commands:

```
CONSTRUCT OBJECTfrog
PLACE OBJECT(150,150)
SHOW OBJECT
```

With these three commands you can tell the computer to construct an object in memory of type "frog," to place that object at a particular coordinate in the Playground window, and to make the object visible to the user.

Seems reasonable enough…

There are a total of 86 commands in FUNdaMENTAL, but don't panic! The language is divided into about a dozen "families" of related commands. In most cases, once you work with one command in a family, it's easy to get to know its siblings and cousins.

## Why Programming?

On the day I wrote my first program, I realized too late that when I called my husband in for the grand performance, all he could see was a little green frog hopping across a white space. He had no way of knowing what that frog meant to me.

### Programming Is Process-Oriented

To me, that little frog represented a process. To get my frog out there hopping, I'd had to first visualize what I wanted my final program to do, and then mentally break it down into the steps I'd need to take to create it. As I went along, I had to make sure that all my commands were in correct sequence, and that I'd accounted for every aspect of my plan.

I freaked out at least 19 times, and more than once thought of throwing the whole computer off the deck. And yet, in finding my way out of various messy tangles, I discovered that my original plan was reshaped and enriched.

By the time I was finally finished, I was already plotting my next program.

Strange as it may seem coming from a recently reformed technophobe, I found the whole process to be somehow familiar. Only after I thrilled to see my froggies hopping did I realize why.

## Programming Promotes Critical Thinking and Creativity

At the heart of the programming process lies something that I recognized in almost any worthwhile endeavor, from writing an organized essay to solving a mathematical proof to building a homemade model of an airplane. It was even present in my effort to understand and bridge the miscommunication between me and my husband.

At the heart of it all is critical thinking and creativity. As a classroom teacher, I've always kept those two biggies foremost in my mind as I planned lessons across the curriculum. In fact, up until recently, I had self-righteously allowed myself to put off bringing technology into the classroom in the name of critical thinking and creativity.

The truth was, most of the "educational" software I'd seen was either a glorified video display or skill and drill. In all cases, the label "INTERACTIVE!" seemed more of a marketing ploy than a truly fair description of the packaged contents. Why should I risk technophobic meltdown in the classroom, I reasoned, if the software doesn't offer anything truly valuable to my students anyway?

So much for that excuse.

Now that I am a FUNdaMENTAL programmer, I'll be bringing programming into my curriculum every chance I get. Although I do many things across the curriculum to promote good thinking skills in my students, I find the benefits of programming to be truly unique.

## Programming Is Structured

On the one hand, there is an enhanced feeling of mental discipline that comes from working within the formal limits of a programming code. And you never have to wait to see if you got it right. You can test-run your program at any time to see how it's coming along. Seeing something unexpected causes you to look closely at your list of commands to see where you went wrong.

As much as it may hurt to admit it, the computer never does anything that you didn't specifically tell it to do! If it messes up, that probably means you did first. This strict structure can indeed be frustrating. But as you start to grasp the patterns of the language, things come

out the way you've planned them more and more often. That's all the reward my students and I ever needed to keep us coming back for more.

### Programming Is Open-Ended

On the other hand, when it comes to what you do with your programming skills, it's almost safe to say the sky's the limit. As programmers advance, they become more and more concerned with issues of style. The challenge lies in discovering ways to achieve complex goals with efficiency and elegance. Animation, video games, strategy games, interactive conversations, simulated calculators, and piano keyboards are just a few of the things that have been created so far with FUNdaMENTAL.

We're eager to find out what you can add to the list!

## Why FUNdaMENTAL?

Now that you think about it, you may have heard of other programming languages besides FUNdaMENTAL. Basic and Logo are two of the most well-known programming languages for beginners. You may know of others. Languages like Pascal, C, C++, Visual Basic, and Java are industrial-strength languages. They are more efficient and run faster than the beginners' breeds. But if you make a mistake, you're on your own and the computer will probably crash, which is never a good sign.

So with all these other programming languages out there, what makes FUNdaMENTAL different?

FUNdaMENTAL is designed so that young and novice learners can create fun, interactive, graphical programs with little difficulty. At the same time, the new programmers learn about what's going on inside the computer when their programs are running.

This is true because, unlike most computer languages for young and novice learners, FUNdaMENTAL is not buried in layers of abstraction. You're not "moving the turtle," or "giving instructions to the robot." FUNdaMENTAL's metaphor for programming a computer is simply that: programming a computer. The user programs in an environment that imitates the internal behavior of a simple computer – one not very different from a basic Macintosh or PC. What is more, the FUNdaMENTAL language models an assembly or machine language, which means that the commands "speak" directly to the different parts of the machine involved in running the program.

If any or all of that flew right over your head, don't worry! The important thing to know is this: FUNdaMENTAL is educational software that is truly interactive, challenging, and, above all, fun. Whether you're a classroom teacher, computer camp counselor, home-schooling provider, or concerned parent, we hope you'll take

this opportunity to discover the rich benefits that FUNdaMENTAL can bring to your learning community.

## What's in This Book?

The bulk of this book contains a tutorial that introduces you to all the elements of the FUNdaMENTAL programming language. It explains how they work and provides hands-on-the-keyboard activities, with step-by-step instructions for starting out on your own programming adventures.

Following the introduction, the main body of the book is organized into three units.

### Unit 1

### Getting Started: Writing Simple Animation Programs

In the first chapter, you'll find a tour of the FUNdaMENTAL interface environment (that's all the stuff you see on the screen when you're working with the software ) There are specific instructions for how to use the mouse and keyboard to write your own programs.

(We assume that you already have some basic computer skills: using the mouse, using menus, clicking and dragging, resizing and moving windows on the screen. If you still haven't mastered these skills, check out the instructions that came with your computer, or grab one of your students and beg them to take pity on a poor, struggling technosaurus. It's fun to learn things that way–don't ask me how I know!)

The subsequent chapters in this unit focus on the basic commands for constructing and manipulating graphical objects with FUNdaMENTAL. By the end of this unit, you should know how to make fun, animated sequences to entertain yourself (at least!) and probably other people, too- especially other programmers!

### Unit 2

#### Creating Interactive Programs and Simple Games

This unit introduces the main commands and skills you need to create programs that are "interactive," which in this case means that the person using your programs will have something to do, as well as to see. By the end of this unit, you should have the tools to make an almost endless variety of simple games and other interactive experiences for your program users (which by this point will likely include a wide audience of adoring fans).

### Unit 3

#### Thinking Like an Expert

This unit presents some of the skills and command groups that can help you design more complex programs and manage them with efficiency and elegance. Where you are at the end of this unit will depend largely upon your background with computers. For some of you it will mean a fairly natural transition into more advanced programming. For others, this unit should provide at least a good understanding and appreciation of what goes into more complex programs (even if you're not yet ready to incorporate the skills and commands into your own programming).

## Featured Sections

Throughout the tutorial, you'll find various special features that help you develop your programming skills.

### LET'S PROGRAM!

These sections appear in each chapter to give you hands-on practice with new commands and skills. You simply cannot learn about programming unless you program!

### Programming Tips and Tricks

Following the Let's Program sections, these Tips provide extra help or suggestions for further thought and activities to help you get the most out of the programming exercises.

## In This Chapter/Unit and Unit Highlights

These sections appear at the beginnings of chapters and at the ends of units. They list the important new commands and skills covered throughout the manual, to help you preview and review as you learn.

## ----- Teacher's Journals -----

Like the one at the start of this manual, these vignettes appear throughout to illustrate and inspire. They capture real student programmers at work with FUNdaMENTAL.

## What's Going On Inside the Computer?

As you go through the tutorial, you will find periodic boxes bearing this title. They will be accompanied by an illustrated model depicting some of the computer's inner workings.

Is this really what's inside your computer?????? Of course not. What's inside your computer is a whole bunch of wires and transistors and other computer guts that you don't need to know about. But that's okay, because you don't have to. It does help to know what all the wires and stuff are up to in there, and that's what the model represents.

It's a "virtual" computer that represents a particular system for processing information, and you can understand the system without knowing how the computer engineers make it all happen with transistors and chips.

You can read these sections before the other parts of the chapter, or after, or both– whatever works best for you. And for you technophobes out there: Please, don't skip them!!! Understanding the way things work is the key to being a good problemsolver. FUNdaMENTAL is designed so that the actual process of using it will enhance your understanding of how the computer works, which of course will better enable you to take advantage of FM's full potential, which in turn will further increase your understanding of what's going on inside the computer, and so on. You'll soon be surprised to find how much you really do understand. Trust me on this one. And, more importantly, trust yourself!

At the end of the book are two appendices designed specially to support you in using FUNdaMENTAL with your students.

### Appendix A: Educator's GrabBag

This section contains four different sections with information and lesson ideas that will help you integrate programming and FUNdaMENTAL into your classroom curriculum.

### Appendix B: More Tools for the FUNdaMENTAL Programmer

Here you'll find a list of all commands with their functions, an extended glossary, and a quick reference of interface features.

## What's the Best Way to Use This Book?

### As an Individual Tutorial

The main purpose of this book is to help whoever reads it become a FUNdaMENTAL programmer. After all, to the extent that programming is an art as well as a science, trying to teach it without being a programmer yourself is a little like giving piano lessons without considering yourself a musician!

(But you don't have to be a master programmer before you can successfully bring programming to your students. I wasn't much past my hopping frog program when I began to teach FUNdaMENTAL. Yet not only was I successful in helping my students become programmers, I even found my own learning and interest in the process increased dramatically once I was working within a community of other new programmers. I didn't have to be an expert; being a professional learner was enough!)

### As a Teaching Resource

In addition to providing you with an individual tutorial, this book is also intended to serve as a teacher's manual. Beyond including the resources in the last section, we have tried our best to structure the tutorial so that it could form the foundation for daily lesson planning. (You may even decide to use portions of the book as student text if you feel it's appropriate.) Of course it will be up to you to determine the particulars of your FM curriculum according to the needs of your students and the requirements of your setting. But

So, now you have all you need to get started. It's time to get hopping!

# UNIT 1
# GETTING STARTED:
## Writing Simple Animation Programs

We all know that there are countless kinds of programs that do millions of different kinds of things, but let's face it: what most people (especially young people) want to know right off is how to make animation.

In chapter 1 you'll take an interface tour, see a demo program, and learn the introductory skills you'll need to write your own programs.

In chapter 2 you'll learn commands for moving and changing objects that are already constructed and visible on the screen.

In chapter 3 you'll learn commands for constructing your own objects according to a particular object design, and for making sure those objects are properly placed and visible to your users.

In chapter 4 you'll learn how to store objects in memory so you can use them again and manipulate them simultaneously with other objects.

In chapter 5 you'll learn commands that tell the computer to repeat portions of the program, so you don't have to write the same commands over and over.

In chapter 6 you'll learn several commands that will give you more creative options and control in writing your animation programs.

In chapter 7 you'll learn how to import your own sounds and graphics to use in the FUNdaMENTAL programming environment.

In chapter 8 you'll learn how to use your customized graphics to design your own blueprints for objects that do tricks when you click on them.

Making animation programs is a great, fun way to start learning about programming. But as you go through the manual please remember that although we may be using one particular type of program for a given lesson, the commands and skills you are learning could be used in any number of kinds of programs, from animated book reports to checkerboards, from word games to jukeboxes. So keep an open mind, and imagine all the things you could make and do with the power tools you're about to acquire.

# CHAPTER 1

## Exploring the Territory and Giving Commands:

## A Guided Tour of the FUNdaMENTAL Programming Environme

## ----- Teacher's Journal -----

It was the first time I'd ever tried to write a program with FUNdaMENTAL, and I felt like I had gotten on the wrong subway train, without even knowing where I was supposed to go in the first place.

"Help!" I cried to no one in particular. Sarah, a kindly programmer working nearby, took pity on me and came over to see what the problem was.

"It says I'm supposed to do something in the Program window," I whined, "But I'm IN the Program window, and I can't find the buttons I'm supposed to use."

"Actually," said Sarah, "you're in the Task window. The Program window is something different. It's hidden right now."

"Hidden? HIDDEN? How'm I supposed to find it if it's hidden?" I said.

"Use the Toolbar" said Sarah with the patience of a Saint.

"Where's the…Oh. There it is," I said, finding the Toolbar. "Now what?"
"Pull down the windows menu," Sarah coached.

"Okay…OOOPS! What did I do now?" I howled, watching the windows on the screen shuffle by rapidly and stop at a blank white window with a frog in the middle.

"You accidentally clicked the Play button. That's okay. You can still see the Toolbar, so use the Windows menu from here."

I gingerly moved the mouse over the Toolbar and pulled down the Windows menu. With the care of a passenger traveling in a foreign land, I chose "Program window" from the list, held my breath, and let go of the mouse.
"There ya go!" said Sarah, as the elusive Program window popped up on the screen.

"AHA!" I said, as if I'd just apprehended a scoundrel. Sarah barely suppressed a little smile and headed back to her desk.

"Don't worry," she said. "Soon you'll know your way around FUNdaMENTAL like you know your way around your own back yard.

# Installing the Program

To begin working with FUNdaMENTAL, you must have the FM software installed on your computer.

Windows 95™
(Note: During the setup, the installer program will ask you to make sure all other applications are closed. Now's a good time to make sure you aren't running any other programs on your computer while you're trying to install FUNdaMENTAL.)

1. Insert the disk labeled FUNdaMENTAL Disk 1 into your PC.
2. In the Start menu, select Run.
3. In the field labeled Open type: a:\setup
4. Follow the on-screen instructions.

Windows 3.x
(Note: During the set-up, the installer program will ask you to make sure all other applications are closed. Now's a good time to make sure you aren't running any other programs on your computer while you're trying to install FUNdaMENTAL.)

1. Insert the disk labeled FUNdaMENTAL Disk 1 into your PC.
2. In the Program Manager, under the File menu, select the option Run.
3. Type: a:\setup
4. Follow the on-screen instructions.

# Let's Learn the Interface Fundamentals

Once you have installed the software on your computer, you are ready to begin exploring the FM programming environment. Opening programs, running programs, entering commands and editing existing program code are the primary skills you need to get started programming with FUNdaMENTAL.

## Launching FM and Opening Existing Programs

1. Launch FUNdaMENTAL. For windows 3.x, do this by double-clicking on the FUNdaMENTAL Executable icon in your Program Manager.

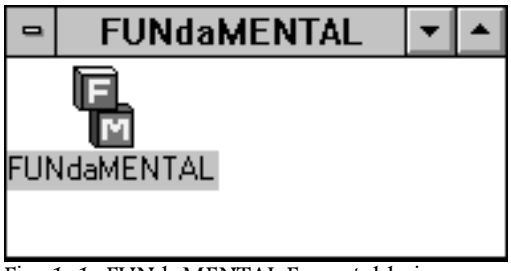



Fig. 1-1 FUNdaMENTAL Executable icon

For Windows 95, click the Start button and move the mouse over Programs. A submenu will pop up. Move the mouse over the FUNdaMENTAL 2.2 icon, and another submenu will appear. Finally, click on the FUNdaMENTAL 2.2 icon that pops up.

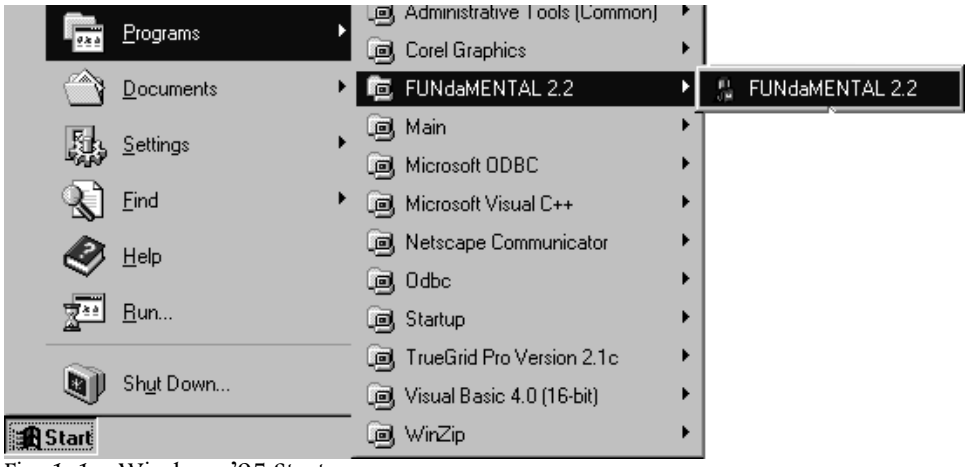



Fig. 1-1a Windows '95 Start menu

This will bring you to the FUNdaMENTAL Welcome window.

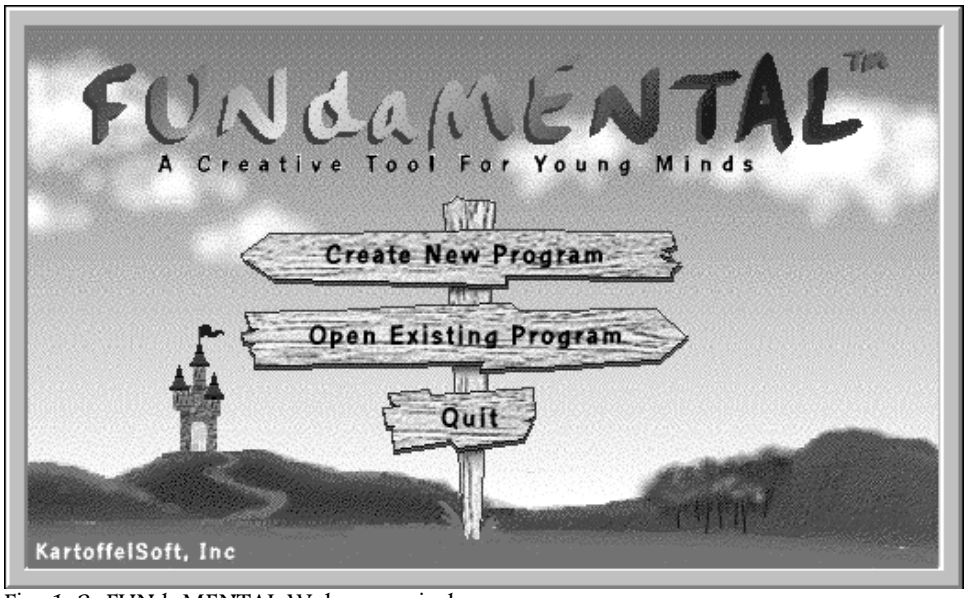



Fig. 1-2 FUNdaMENTAL Welcome window.

The Welcome window gives you three choices. You can create new programs, open existing programs, or quit FUNdaMENTAL.

2. Open Existing Programs. Do this by clicking once on the Open Existing Programs button. This will bring you to the Open Program directories window.

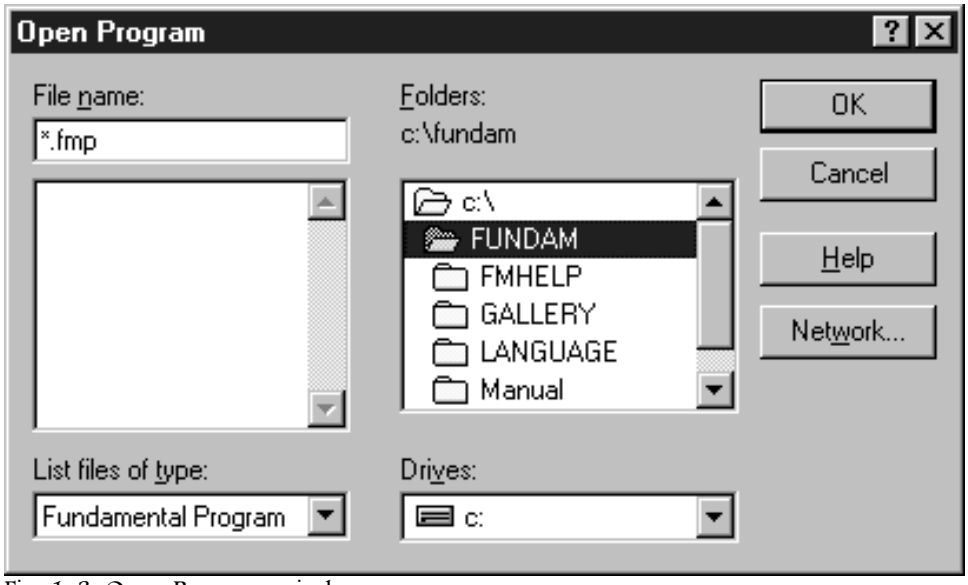



Fig. 1-3 Open Program window

3. Starting from c:\, open the folder marked FUNDAM. It may already be open, but it's best to be ready to start "from the top" when you set out to navigate through directories. If it's not open, double-click on the word FUNDAM which appears next to the little folder icon in the list of things in c:\.

4. Open the folder marked Manual. Do this by double-clicking on it in the directories list.

5. Open the folder marked Practice. Double-click again.

6. Open the folder marked Demo1. Another double-click. The file name "Demo1.fmp" will now appear in the files list on the left. (The ".fmp" stands for FUNdaMENTAL program)

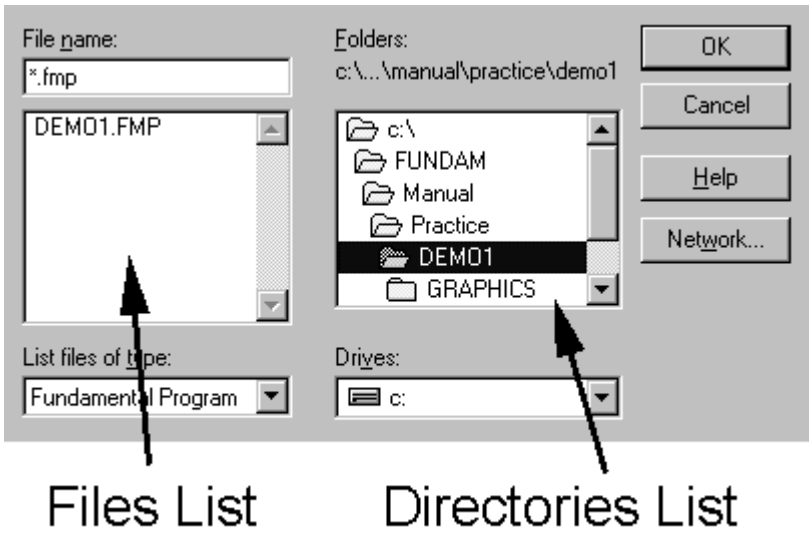



Fig. 1-4 Demo1.fmp in file list, ready to open

7. Open this program. Do this one of two ways: 1)click once on the file name, Demo1.fmp in the files list, and then click the OK button on the upper right side of the window or 2) double-click on the file name Demo1.fmp in the files list.

8. Don't try this now, but…to close one FUNdaMENTAL program and open another without quitting FM, use the File menu on the Toolbar and select Close Program. This will bring you back to the Welcome window, from which point you can begin again by clicking on the Open Existing Programs button. For now, though, stay in Demo1.fmp. Goodness knows we worked hard enough to get here; let's stay awhile!

   You should now be looking at the *main* Task window for Demo1.fmp.

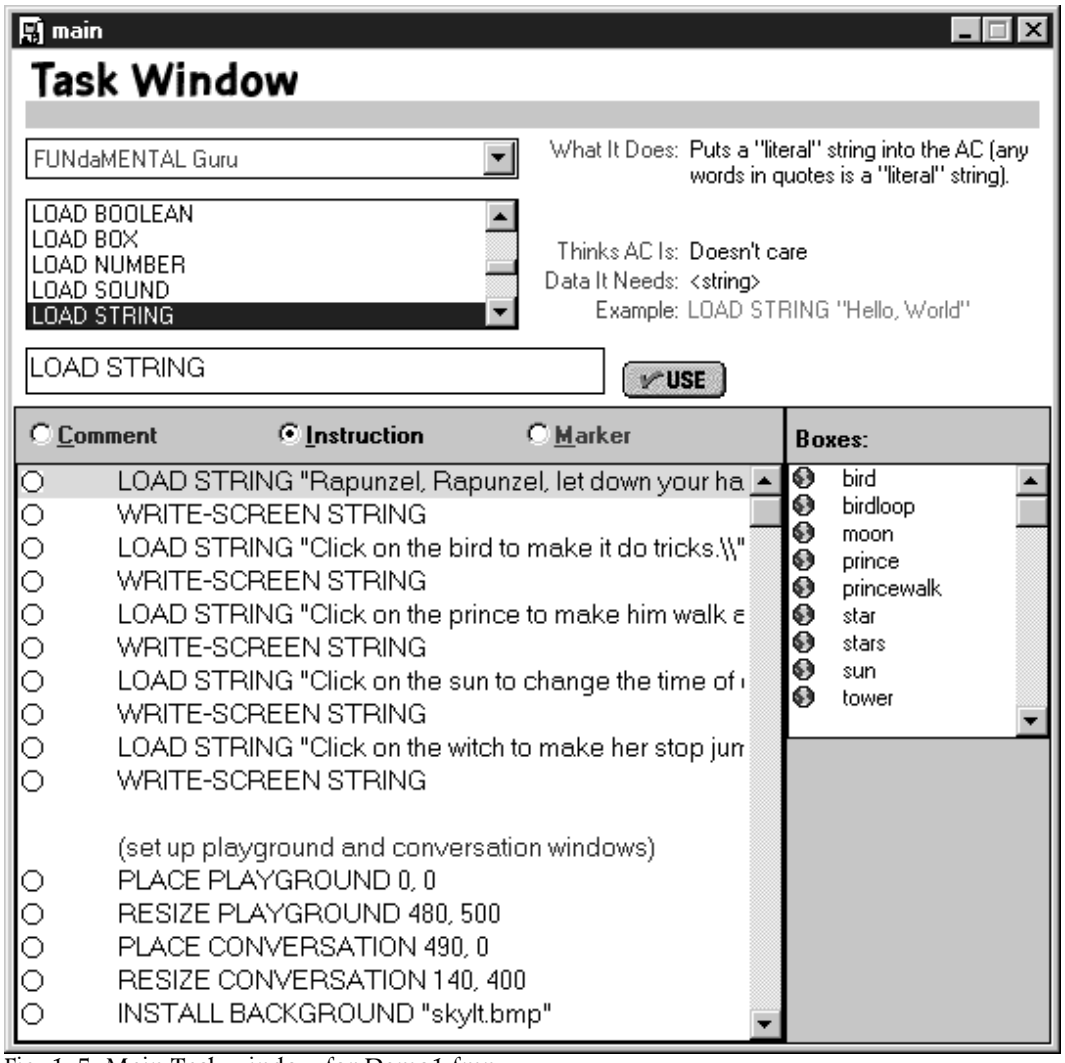


Fig. 1-5 Main Task window for Demo1.fmp

This is the place where the commands are entered to make a program. Because you have chosen to open an existing program, you should see some text in the Instruction list. Those are the FUNdaMENTAL commands that are responsible for making the program run the way it does. (Note that if back at the Welcome window you had chosen Create a New Program, you'd see the same Task window, but the Instruction list would be blank. It would be up to you to fill in all the commands to make the program.)

You'll get a chance to run this program in just a minute. In other words, you'll get a chance to see and hear and interact with everything that happens when the computer follows all the instructions in the commands list. But before you do, take a minute to look at the commands (collectively called "the code") for this program…*Stop and predict*:

- What, if anything, do you think you might see when this program is run?

- What, if anything, do you think you might hear when this program is run?
- What other things might happen when this program is run?

Take a minute to record your predictions in a couple of sentences, or even make a quick sketch.

(This is not a quiz! "I haven't the foggiest idea!" is a perfectly acceptable prediction at this point in your programming career…but I bet if you give yourself a minute to examine the instructions in the Instructions list, you will find that you do, indeed, have at least a nice little foggy idea about what you're about to see.)

## Running FUNdaMENTAL Programs

1. Find the Toolbar. It looks like this:

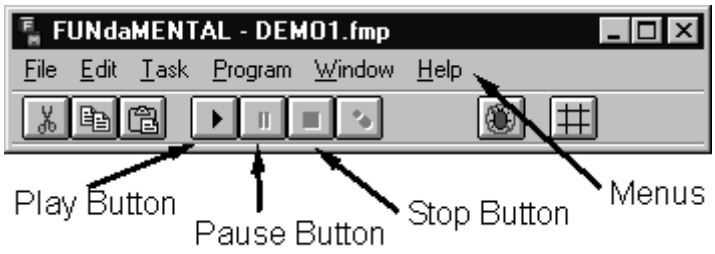



Fig. 1-6 FUNdaMENTAL Toolbar

It should be near the top of your screen. Since it's the central control panel for FUNdaMENTAL, the Toolbar will always stay "on top of the pile" of all the windows on your computer desktop. That means it will always be visible. But it's a draggable window, so you can move it anywhere you choose.

1. Click once on the Play button. After a little more shuffling of windows, the program will begin to run. (You may want to drag the Toolbar out of the way in order to see all the action.)

2. Watch and play with the program. Rapunzel and her prince are foiling the Witch, and it's all happening in the Playground window. Read and follow the instructions that are displayed in the Conversation window. When you ran the program, there were things going on in two different places. The Playground window, which displays the animation, is where most of the fun is! The Conversation window displays text and receives keyboard input so that the program and the user can exchange instructions, insults, or pleasantries, as the programmer sees fit. To stop the program, click on the witch, or on the Stop button on the Toolbar.

3. Return to the main Task window. Do this by pulling down the Window menu on the Toolbar and dragging the mouse to select

the last item in the list, Main. You should now see the main Task window for this program.

## Entering Commands

1. Find the end of the Instruction list. Do this by using the scroll bar to the right of the Instruction list in the Task window. Look for the command EXIT PROGRAM

2. Get ready to enter an instruction. Click on the list, in the space above the second to last command, SLEEP MAIN. The blue highlight indicates an open space, ready to receive a command.

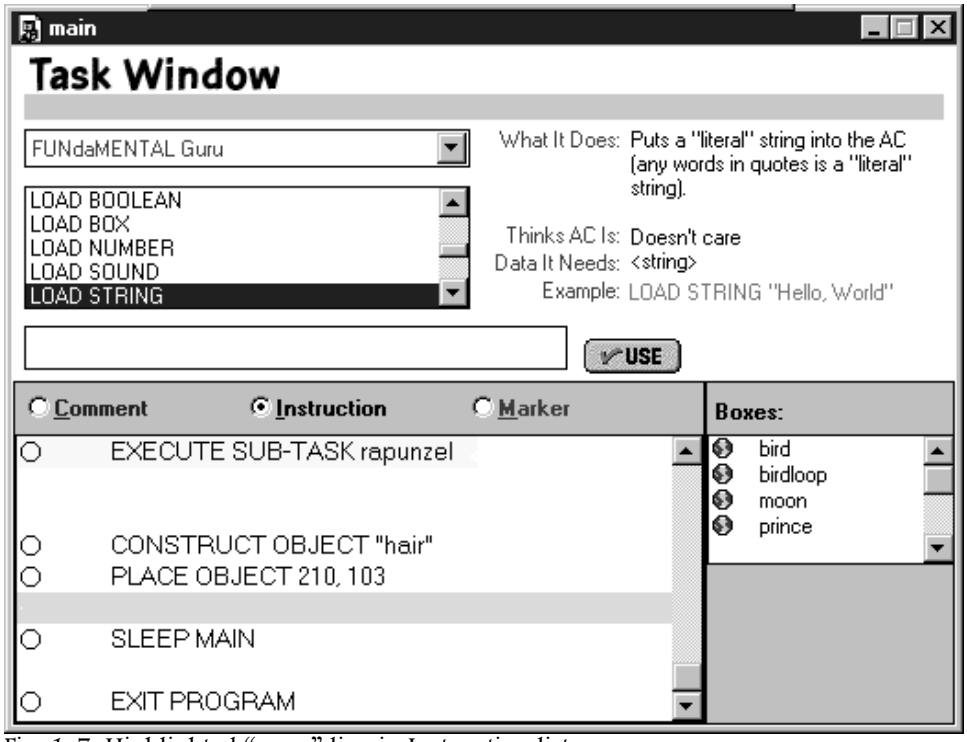


Fig. 1-7 Highlighted "open" line in Instruction list

Notice that the two instructions above your empty line tell the computer to construct an object of type "hair," and to place it on the Playground. So far, though, it's invisible.

3. Tell the computer to show this object. Although we've made a space in the Instruction list, this is not where we actually enter the command. Above the Instruction list, there are three radio buttons marked Comment, Instruction, and Marker which allow you to select your code type. Click on the Instruction radio button so the computer knows what kind of information you're about to give. Now move your cursor up to the text-entry field above and select the command SHOW OBJECT in one of two ways: 1)begin typing it into the text-entry field or 2)scroll through the All Commands list above the text-entry field, and

click on the command you wish to use. It will then appear in the text-entry field.

4. Confirm your command choice. Once you see the command SHOW OBJECT in the text-entry field, click on the Use button to the right, and the command will appear in the highlighted space in your Instruction list. You can achieve the same thing by double-clicking on the command where it appears in the scrolling All Commands list or by pressing the Enter key on your keyboard.

5. Run the program to see how it's changed with the inclusion of your new nstruction. Click on the Play button on the Toolbar to do this. (Now *that's* Rapunzel). When you're ready, click on the Stop button on the Toolbar. Now use the Window menu on the Toolbar, and select Main to return to the ain Task window so we can edit the code for this program.

## Editing Existing Code

Many times, when you want to add a new instruction, as we just did in the above exercise, there isn't an empty space in the Instruction list waiting conveniently where we need it. Instead, we need to insert a blank line between two adjacent instructions. Just as often, we discover that a particular instruction in the list needs to be deleted altogether. We'll learn how to insert and delete lines in the Instruction list now.

1. Insert a blank line in the list. Move the cursor over the last command in the Instruction list, EXIT PROGRAM. Click on it. It should now be highlighted in blue. (Please note that when you want to highlight a line currently occupied by an instruction, click in the middle of the line, and not on the little open dot to its left. If you click on that dot, it will turn red, like a little stop sign. A red dot next to an instruction causes the computer to stop on that instruction when executing a program. This is a tool programmers use when they're trying to find an error, or "bug," in a program. If you happen to click on the dot and turn it red by accident, don't panic; just click it again, and it will return to its original color.) Once you've highlighted the line where you want to make your insert, use the Edit menu on the Toolbar, and select Insert Line. The command EXIT PROGRAM will move down a space, and the blue highlight will remain on the new, blank line, ready to receive a new Instruction. (Notice that you can achieve the same effect from the keyboard, using a Control Key-I combination.)

2. Delete a line from the list. Let's give Rapunzel a haircut. Find the new Instruction that you entered during the last exercise: SHOW OBJECT. It should be the last instruction before EXIT PROGRAM. Click on it once to highlight it. Then, use the Edit menu on the Toolbar, and select, Delete Line. The SHOW OBJECT

command will be deleted from the Instruction list. (Notice you can achieve the same effect from the keyboard by using a Control Key-K combination.)

3. Run the program to see how the changes you've made affect it.

## Let's Explore a Little Further

You've already acquainted yourself with the majority of important interface features in FM. There are, however, a few other windows that you'll have more and more occasion to use as your programming skills increase. We'll visit them now to take a look and learn more about what they do and how to use them later on in the book.

### Using the Window Menu

All of the primary interface destinations in FM can be reached from the Window menu on the Toolbar.

1. Find the Program window. In the Window menu on the Toolbar, select, Program window. It looks like this:

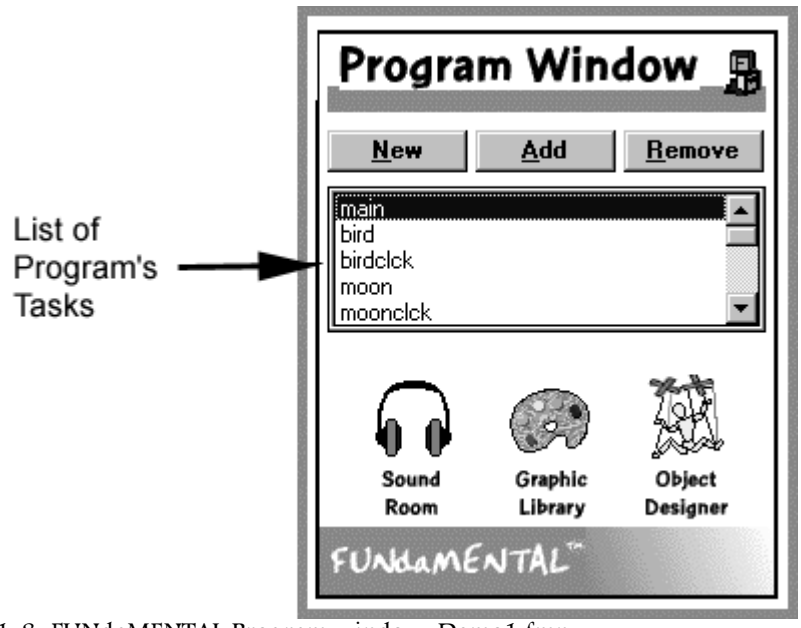



Fig. 1-8 FUNdaMENTAL Program window, Demo1.fmp

Don't confuse this with the Task window, which holds the list of your instructions for the computer. Although it is possible to have a program with just one "main" list of commands, most programs are made up of a bunch smaller sub-tasks, like the ones you see listed here in the Program window. Each one is like a miniprogram in itself, which is responsible for a particular aspect of the program's total action. Think of this Program window as the directory, or table of contents, to all the components, or sub-tasks, of your program. From the Program window, you can get

into any of the sub-tasks that make up the entire program. Double-click on the close box in the upper right-hand corner of the Program window once you've taken a look.

1. Find the Object Designer. From the Toolbar window menu, select the Object Designer. It looks like this:

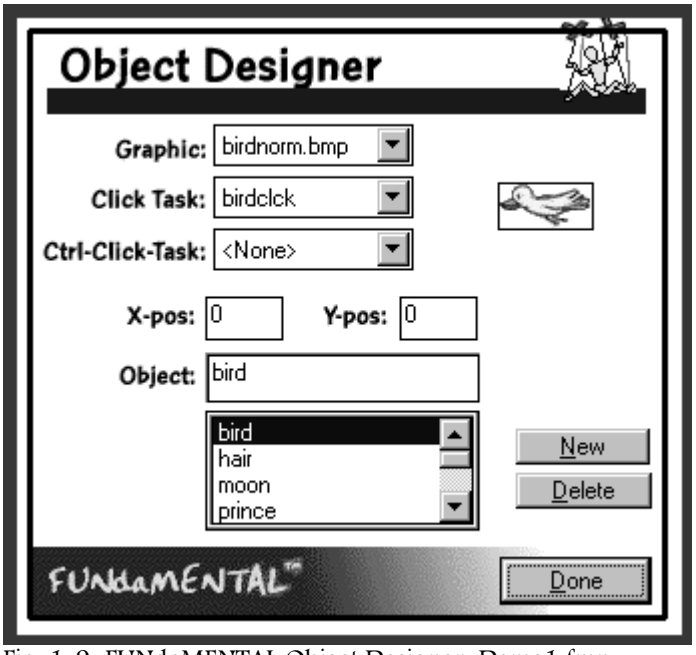



Fig. 1-9 FUNdaMENTAL Object Designer, Demo1.fmp

Here is where it possible to design the objects for your programs by combining graphics with other information. Best to leave all the buttons alone for now. We'll learn to use them later. Please note: *The Object Designer is not where you give program instructions to the computer*. Think of it as a place where you assemble the supplies that the computer will need to follow the program instructions which you enter in the Task window. Click on the Done button when you have finished taking a look at the Object Designer.

2. Find the Graphics Library. From the Window menu on the Toolbar, select "Graphics Library."

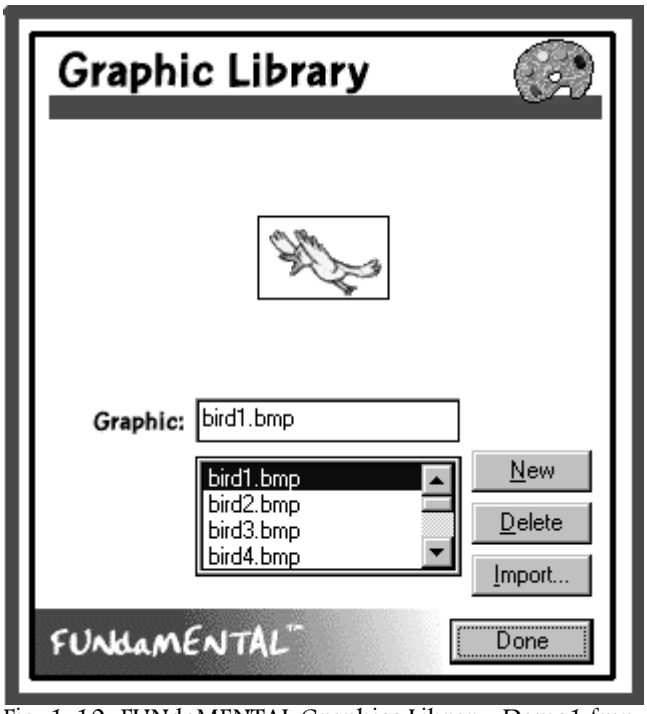



Fig. 1-10 FUNdaMENTAL Graphics Library, Demo1.fmp

This is where you can import and store the graphics you need for your object designs and program backgrounds. Again, just use the scroll bar to take a look, but leave the buttons alone for now. And remember: *Working with graphics in the Graphics Library is not programming!* It's like going to the store to buy the supplies you'll need to have on hand for the computer to follow the instructions you give in the Task window. Click the Done button when you are finished taking a look.

3. Find the Sound Room. In the Toolbar Window menu, select the Sound Room. It looks like this:

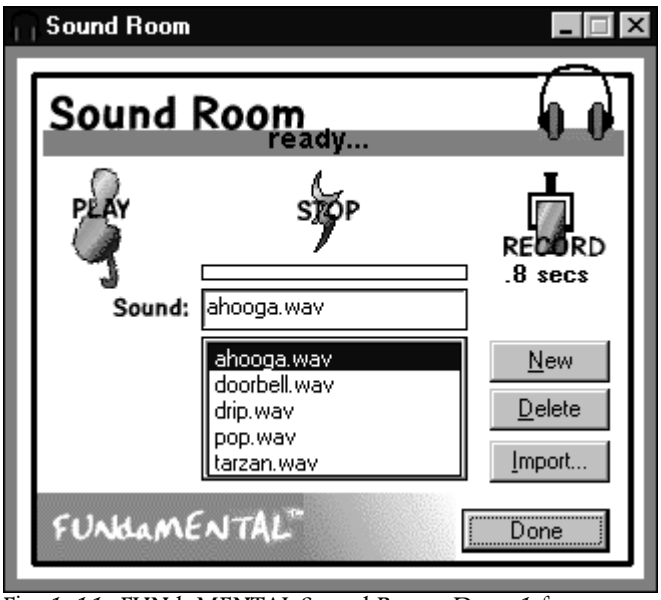



Fig. 1-11 FUNdaMENTAL Sound Room, Demo1.fmp

Here is where you can record, import, and store the sounds you tell the computer to use in your program. Take a look at the list, and then click the Done button.

If it all seems a little overwhelming, don't worry. We'll be reviewing all this information as we go, and before you know it, (just as my programming friend Sarah promised me not too long ago) you'll know your way around this interface like you know your way around your own neighborhood.

So, now that you've had a chance to survey the territory and get a feel for writing commands, it's time to get to the fun part! Let's program!

## Programming Tips and Tricks

Many people find the keyboard faster than the mouse for getting around the interface and performing basic functions such as saving work on the computer. FUNdaMENTAL has a number of useful keyboard shortcuts. You can see them displayed next to certain items in the menus on the Toolbar. Most of them work by holding down the Control Key, while pressing another key on the keyboard. Here are a few you might find useful as you begin learning to program:

Control Key-S: Save (It's VERY IMPORTANT to remember to save your work frequently while your working on a program!)
Control Key-I: Inserts a line in the Instruction list
Control Key-D: Deletes a line in the Instruction list
Control Key-W: Closes the active *window* (not to be confused with closing an FM *program*...for that, you have to use the mouse and the File menu.)
Control Key-G: "Go": like clicking the Play button on the Toolbar
Control Key-J: "Abort": like clicking the Stop button on the Toolbar
Control Key-Q: Quits FUNdaMENTAL
Control Key-P: Prints the program code in an active Task window
Enter key: Can be used instead of clicking on the Use and Done buttons in the FM interface
F1 key: Press this key after clicking once on an FM interface feature, and you will see an on-line Help screen with information about that feature.

# CHAPTER 2
## Moving And Changing Objects



Commands Introduced:

## ----- Teacher's Journal -----

It was a half-hour after the end of the period on a warm summer school day, and LaToya wouldn't leave the classroom. All the other kids in the class were long gone to lunch and recess, but LaToya was oblivious to the sounds of calling voices and running feet in the hallways.

"Just one more command! ONE more…I'm about to get it. Pleeeease!" LaToya pleaded, never taking her eyes off the computer. I made a half-hearted show of looking at my watch, but in truth, I was just as excited as she was. Before this, I had been nearly ready to give up on her. Deeply distracted by grave family troubles, LaToya had always been better at making excuses than making an effort. But not today.

She tapped away at the keys, adding new instructions to her program task list, and then clicked on the button to give it a test run. In a moment, her own simple drawing of an Olympic diving platform appeared on the screen, with a stick-figure diver waiting at the bottom of the ladder. LaToya clasped her hands under her chin in suspenseful anticipation and watched as the computer followed the instructions she had given it.

First, the little diver stepped forward and climbed the ladder to stand erect at the back end of the platform. LaToya applauded this small feat with delight, even though it wasn't the first time she'd seen it. The real test was in what followed.

As we both held our breath, the little stick figure raised one knee after the other and marched resolutely toward the end of the platform. Our cheers turned to laughing groans as the diver took three or four extra steps off the end of the board and levitated in the air above the water.

"Oh NO!" LaToya said, slapping her forehead melodramatically. She quickly switched back to the program task and scanned through her list of commands to find the bug in her program.

"LaToya," I began, looking at my watch with the beginning of real concern that I might actually miss an afternoon meeting.

"Just ONE more, Jan! JUST ONE MORE.…"

## Let's Get Moving!

What is animation all about, anyway? It's about moving objects around on a screen or, in this case, in the Playground window. LaToya, for example, was trying to move her diver object up the ladder and across the diving platform. Here's the primary command she was using to do it:

✔ MOVE OBJECT

This command does exactly what it says. It moves a specified object according to your instructions.

Whenever you use this command, it automatically brings up a new interface feature, the Data Wizard.

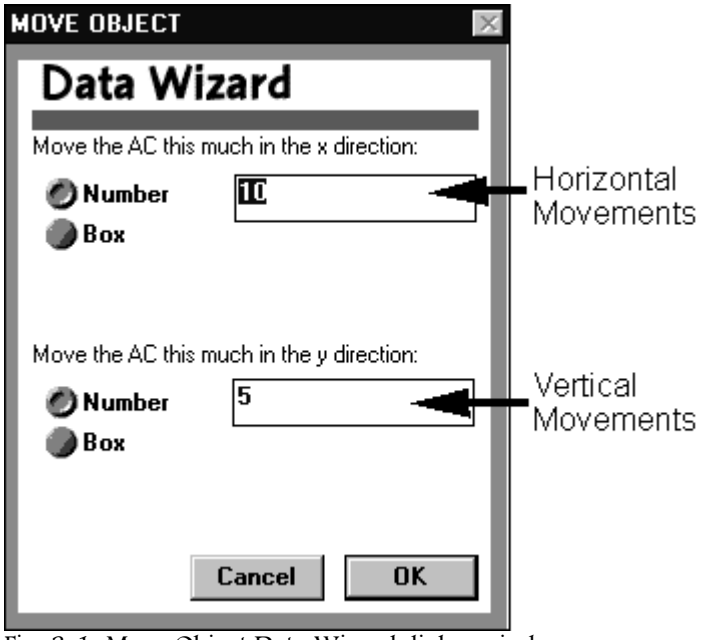



Fig. 2-1 Move Object Data Wizard dialog window

This is where you fill in the details that complete your instructions to the computer. In this case, when you say "Hop!" the computer will literally ask you "How high?"

## The Grid

The computer automatically thinks of the Playground window as an x/y coordinate grid, with the point (0,0) in the upper left-hand corner. (Notice that the programming grid is an "upside-down" version of the conventional Cartesian grid.) When you tell the computer to MOVE OBJECT, the Data Wizard will ask you for an x-distance (horizontal) and a y-distance (vertical) by opening up a little dialog box that looks like this on your screen.


Fig. 2-1a Grid button on Toolbar

You can type numbers straight into the text boxes for each direction. For movements away from the point (0,0) –which in this case is at the upper-left corner of the Playground –use positive numbers, and for movements back toward (0,0) use negative numbers. In other words, negative numbers in the x field will cause the object to move back toward the left edge of the Playground. Negative numbers in the y field will cause the object to move *up,* toward the top of the Playground.

(You may have noticed that the Data Wizard offers the option of using a box instead of a number. This allows for the use of variables in more complex programming–to be covered in later chapters.)

Until you get a good feel for the dimensions of the grid, it helps to see it. To see the grid during a test run of your program, just click on the Grid button on the Toolbar. The grid will appear superimposed on the Playground window as your program runs.

Note that while your program is running, you cannot see the Task window that holds your list of commands. If you want to see the Playground grid while you're working in the Task window, go to the Window menu on the Toolbar and select the Playground window. If you have run the program before doing this, then the Playground will appear with your program graphics in it. If you do this before running your program, then the Playground will appear blank. (Either way, you may need to drag the Playground out of the way to be able to see the Instruction list.) Once you see the Playground window, click on the Grid button and…voila! The grid appears before your very eyes. To make the grid disappear click the Grid button again. (To make the entire Playground disappear when you're done, click on either the little box marked with an x in its upper right-hand corner for Windows 95, or on the little box marked with a - in its upper left-hand corner for Windows 3.1.)

## Get Ready to Program

Time to try all this out!

PLEASE NOTE: For your first practice experiences, you'll be adding things to programs that already exist so you can focus your attention on the new commands and skills at hand. Before long you'll know enough to start your own programs from scratch.

## LET'S PROGRAM!

1. Start at the FM Welcome window. If you're already inside a FUNdaMENTAL program, then you need to go to the File Menu on the Toolbar and choose Close Program. That will bring you back to the Welcome window. From there, you can get to another place in FM. If you're starting a new session with this exercise, launch FUNdaMENTAL, and that will bring you to the FM Welcome window.

2. Click on Open Existing Programs, and navigate to Program1.fmp. In the Open Program window, double -click on the following folders: c:\ then Fundam then Manual, then Practice, then Program1.

3. Open the program. You should now see Program1.fmp in the files list to the left. Double-click on that, or click once, and then click the OK button. After passing back by the Welcome window, you'll see a simple program code written out in the Task window.

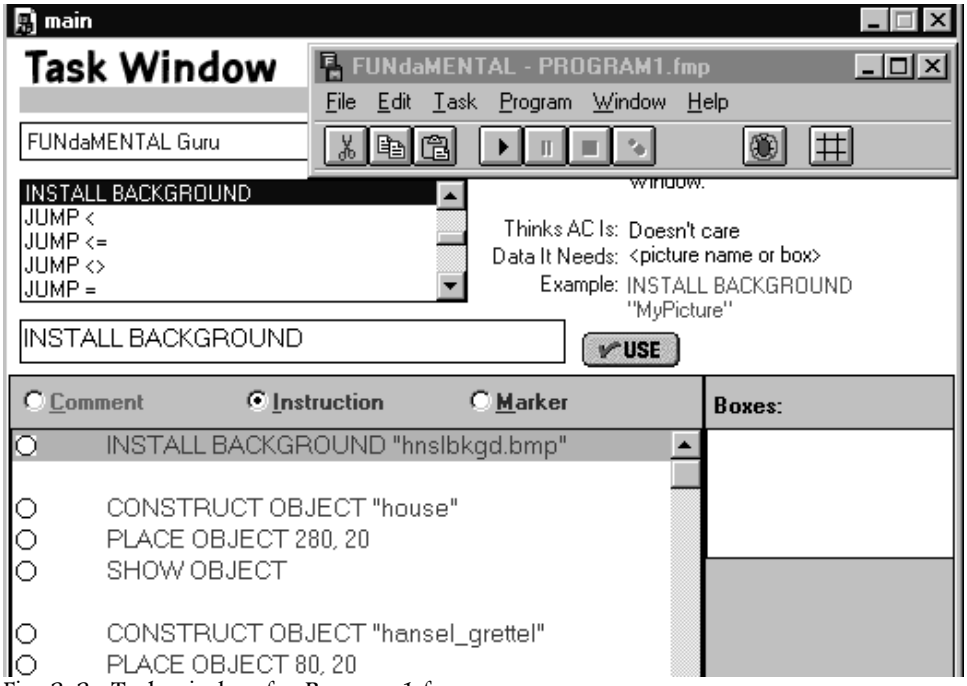


Fig. 2-2 Task window for Program1.fmp

4. Take a look at the code and see if you can make some predictions about what might happen when you run this program. What do you think the program does so far? What do you think it looks like and sounds like?

5. Run the program. On the Toolbar, click the Play button and see what this program does so far. Return to the Task window, and accomplish your mission: Use the MOVE OBJECT command to make Hansel and Gretel walk toward the witch's house!

6. Insert a few lines in the Instruction list. Click the mouse in the Instruction list on the line just above the command EXIT PROGRAM. Use Control Key-I on your keyboard three or four times, and the EXIT PROGRAM command will move down. Highlight the first blank line under under these instructions:

   CONSTRUCT OBJECT "hansel-gretel"
   PLACE OBJECT 80, 20
   SHOW OBJECT

7. Use MOVE OBJECT several times, with different number inputs each time. Make sure that the button next to the word "Instruction" is selected. Click the mouse in the text-entry field below the list of all commands. Now, begin to type in the command MOVE OBJECT until it appears in the field. Click on the Use button. The MOVE OBJECT Data Wizard dialog will appear. Now type in the distance in each direction you want the object to move by using the data-entry fields in the Data Wizard. Click on the OK button to enter the command into the Instruction list. Your inserted lines of code may look something like this:

   MOVE OBJECT(10,10)
   MOVE OBJECT(10,-10)…

When you're finished entering three or four new lines of MOVE OBJECT code, click the Play button on the Toolbar to see your program run. To go back to the Task window and make additions or changes to your program code, you can either click on any portion of the Task window that may be visible, or you can use the Window menu on the Toolbar and select the Task window to bring it back up to the top of the pile. If you want to change any of the number inputs for your MOVE OBJECT commands, simply click on the command in the list and then click on the Use button. The Data Wizard dialog will appear, and you can change the numbers for horizontal and vertical movements for that command.

## Programming Tips and Tricks

It's possible to get your objects from point A to point B with one MOVE OBJECT command, but you'll have smoother animation if you string together several of these instructions…Take your time, and see how creative you can get with this one powerful command. Can you make Hansel and Gretel do zigzags? Loopdy-Loops?

Don't forget to use the Grid button on the Toolbar if you need help planning your next move.

If you're ending your session, don't forget to save your program by selecting Save from the File menu on the Toolbar, or by holding down the Control Key and then pushing the "S" key on your keyboard.

Congratulations! You just wrote your first program! (Remember, even though a few commands were already there, nothing happened until you finished it off.) Stay at this same program. We're going to add some fun stuff to it.

Here's another great command for animating objects:

✔ MORPH OBJECT

If you're a Kafka fan, you may already have a hunch what this command does....if you aren't, we'll just go ahead and tell you. It changes the appearance of the object that the computer is "thinking of" by changing the graphic that belongs to the object.

Aha! I bet you thought an object *was* a graphic. But actually, they're two different things. A graphic is only part of an object–the part that determines how the object looks. Working with the MORPH OBJECT command will be the first step in helping you develop a feeling for the distinction. There are a couple of things you should know before you start "morphing" objects.

## Available Graphics

Whenever you use MORPH OBJECT, it automatically brings up the Data Wizard. This time you should see a list of the available graphics in the Graphics Library.

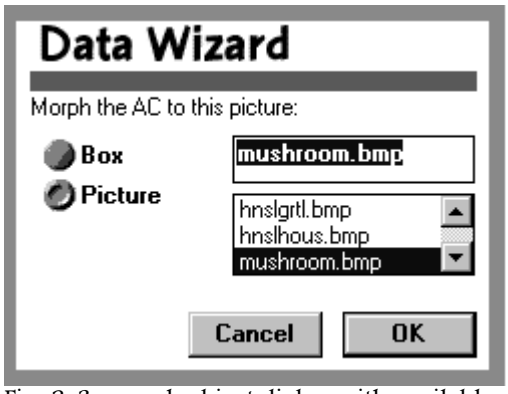



Fig. 2-3 morph object dialog with available graphics for Program1.fmp

If the object you want to change looks like a pair of children, for example, you might want to morph it into the graphic called gingerbrd.bmp (Can you guess the meaning of this abbreviation? If you wish to actually see the graphics at your disposal, you can do so by going to the Graphics Library itself.

Get there by pulling down the Window menu on the Toolbar, and selecting Graphics Library. Once you're in the Graphics Library, you can see the pictures associated with the names displayed in the Data Wizard dialog box. Click on each graphic name in the list to see the graphic that belongs to it, and to familiarize yourself with the names. Click the Done button to get back where you were before.

You can choose the name of the graphic you want your object to morph into by clicking on it in the list that appears in the Data Wizard.

Try using this command to add a new twist to your MOVE program.

## LET'S PROGRAM!

You need to be back inside Program1.fmp for this exercise. If you're not there already, launch the FM application and click on the Open Existing Programs button at the Welcome window. In the Open Program window that appears, double-click on the Fundam folder under c:\ in the directories list. Double-click again on the following: Manual, Practice, Program1. Double-click Program1.fmp where it appears on the left in the File Name list. That should get you where you need to be.

1. Find a good place to insert a MORPH OBJECT command. Look over the code for your MOVE OBJECT program, in which Hansel and Gretel approach the witch's house. Decide on a good place for Hansel and Gretel to get morphed into something else...maybe two gingerbread kids? Two toadstools?

2. Insert a space between existing commands, if necessary. (You want them to be transformed in the middle of the action. Do this by first clicking on the command that currently occupies the desired space. Then select Insert Line from the Edit enu at the top of the screen, or hold down the Control Key and press "i" on your keyboard. That will make a blank space in the middle of the existing lines of code.)

3. Use the command MORPH OBJECT, and give Hansel and Gretel a whole new look! Once you type the command in the text-entry field and click on the Use button, the Data Wizard dialog will appear with a list of available graphics. Choose the name of the new graphic from the list in the Data Wizard by double-clicking on it (try toadstl.bmp or gingerbd.bmp). Click the OK button in the Data dialog to enter the command into the Instruction list.

## Programming Tips and Tricks!

You can see here that objects are different from their graphics. Remember that even after you have morphed Hansel and Gretel into mushrooms, as an object, they are still called hnslgrtl. If you want the mushroom to keep moving around, you still use the same instruction as before the transformation: MOVE OBJECThnslgrtl

A sample solution to this assignment can be found in the folder marked Examples in the list that appears when you double-click on the Fundam folder in Directories list in the Open Program window.

Get there by selecting Close Program from the File menu on the Toolbar, and then clicking the Open Existing Programs button at the Welcome window. This is a good place to visit if you're stumped on an exercise, or if you're just interested in seeing how another programmer would do it.

## Flip-Book Fun: Coordinating Move and Morph

Do you remember playing with flip books when you were a kid? You hold the book in one hand and flip the pages past your other thumb to see Mickey and Minnie do a waltz or Donald Duck take a fall. Each page shows the same picture with a slight variation in the placement of the figures so that the images blend to create the illusion of movement when the pages are flipped.

Can you guess where we're heading with this? As we've already seen, you can morph an object into an entirely different thing (and make a caterpillar turn into a butterfly, or a cat turn into an alien). But you can also morph an object to show the *same thing in a different position*. If done repeatedly, this can achieve a fun flip-

book effect in your animated programs. Let's try this now. Use the File menu in the top left corner of your screen, and select Close Program. This will close the program you've been working on and bring you back to the Welcome window.

## LET'S PROGRAM!

1. Open Program2.fmp. At the Welcome window, click on the Open Existing Programs button. In the Open Program window, double-click on the folder marked Fundam under c:\ in the directories list, and then navigate through Manual and Practice folders to find the folder for Program2. Open that program by double-clicking Program2.fmp where it appears in the Files list in the Open Program window.

2. Before you click on the Play button on the Toolbar, look at the code and make some predictions about what you're about to see. Now test run the program. Not much happening, is there? But we won't let that be true for long! Time to accomplish your mission: Make the frog do jumping jacks and then hop across the screen!

3. Visit the Graphics Library to see what you've got to work with. Start by using the Window menu on the Toolbar, andselect the Graphics Library so you can get familiar with the available graphics. Click on the different graphic names in the list to see the different pictures available for you to use in this program. Notice that they are all pictures of the same frog in different positions. Click Done when you are finished viewing the graphics.

4. In the Task window, use the command MORPH OBJECT. In the Instruction list, highlight the line under the command SHOW OBJECT. Now click in the text-entry field. Type or select MORPH OBJECT and click the Use button. When the Data Wizard comes up, click on the button next to the word "picture," and then choose a picture of the frog in a different position from the list of available graphics. Click Done to enter the instruction into the list under the existing code.

5. Play around with MOVE OBJECT and MORPH OBJECT commands to animate your frog. Just use MORPH commands for several lines to get the jumping-jacks effect. Then intersperse MOVE commands to get him hopping...

(Remember, your family may not be so impressed with your programs yet, but you certainly should be!)

## Programming Tips and Tricks

Take your time and play around. Keep your program MOVIN' and MORPHIN' in as many different ways as you can think of! A sample solution for this, as well as for all other assignments, can be found inside the folder marked Examples in the list that appears when you double-click on the Fundam folder in the Directory list in the Open Program window. Get there by selecting Close Program from the File menu on the Toolbar, and then clicking the Open Existing Programs button at the Welcome window.

# CHAPTER 3

## Constructing Objects
## ...and getting them out there
## where your audience can see them!



Commands Introduced:

## ----- Teacher's Journal -----

"Where is it?" Shane asked.

"Where's what?" I asked.

"The object," Shane said, tipping her chair waaaay back to make sure I knew she didn't buy any of this. "I told the computer, 'CONSTRUCT OBJECT,' but I don't see any object there when I run it." She clicked the Play button on the Toolbar to demonstrate her action-free program. Sure enough, nothing happened.

"See?" she said, content in confirming her own grouchy outlook on the whole project. "You said I was the boss and the computer would do whatever I tell it to do, but it didn't construct any object."

"Oh yes," I said, "it did."

"Then *where is it???"* she asked, giving in to exasperation.

"Right here," I said, tapping the casing on the computer, "inside the computer's itty-bitty brain."

"But I want it to be right HERE!" Shane said, flicking the screen.

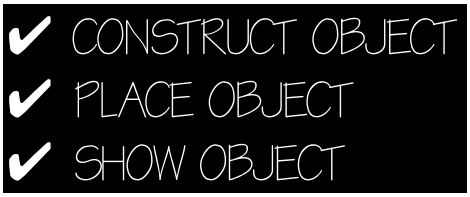
"Then you need to *tell* the computer that," I said.

"You mean I have to tell it to show the stupid thing even though I already told it to build it?"

"You're the boss!" I said.

## Where Do These Objects Come From?

So far, you've been working with objects that have already been constructed for you. In the practice programs you've used so far, you may have noticed these three commands in the Task window:




The first one tells the computer to construct an object according to a specific blueprint in the Object Designer. You can make a whole bunch of the same kind of object, or a set of different objects.

Once you've asked the computer to construct an object, you have to tell the computer where to place the object in the Playground window, and also remind the computer to show the object to your audience.

It may seem odd at first to have to tell the computer anything more than "construct object," but that's all in the day's work of the programmer. The big secret here is that the computer is really, REALLY stupid. You have to tell it eeeeeeeeeverything!

## The Object Designer

When you use the construct object command, the Data Wizard shows you a list of names belonging to the object blueprints that are currently in the Object Designer.

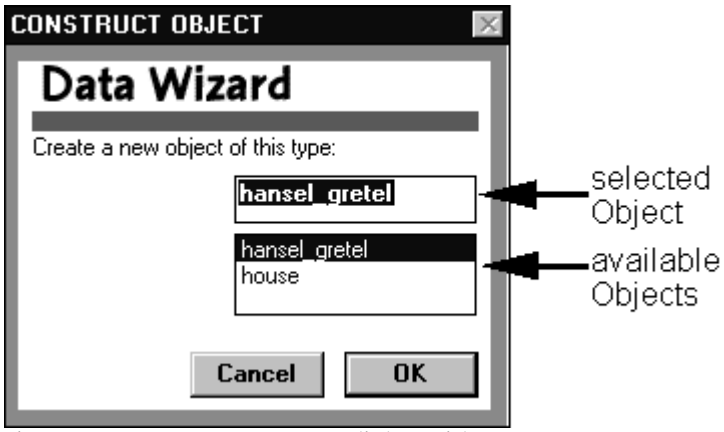



Fig. 3-1 CONSTRUCT OBJECT dialog with available object designs for Program1.fmp

PLEASE NOTE: Although this list may look somewhat similar to the list that corresponds to the contents of the Graphic Library, objects

and graphics are completely different things. It may help you to think of the object as the character in your movie and the graphic as the costume, make-up, or even special effect that determines the way the character appears to the audience at any given time. An object called "Alien" may start out wearing a "costume" called "alien," but may easily morph into another costume, say, "littlegirl" or "alienpuddle", by the end of the program. Despite the "costume change," the object (or character) is still the same Alien.

When you first start working with CONSTRUCT OBJECT, you'll be working with object blueprints that we've already designed for you. You can see the actual object designs by choosing Object Designer in the Window menu on the Toolbar.

When you use PLACE OBJECT, the Data Wizard will appear again.

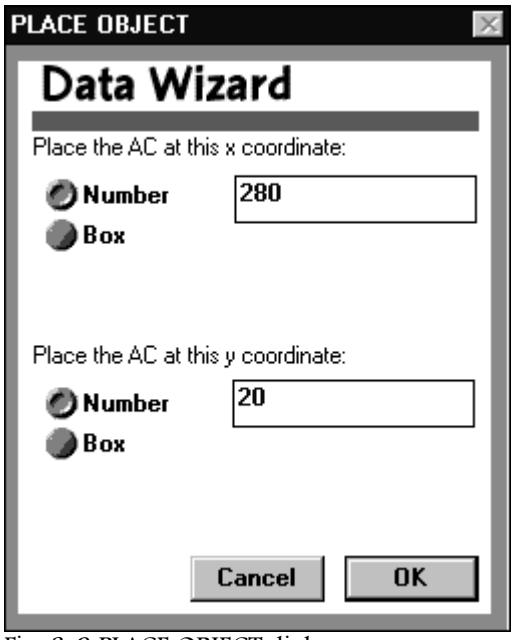


Fig. 3-2 PLACE OBJECT dialog

You use it here pretty much the same way you use it for the MOVE OBJECT command, only this time the numbers signify particular coordinates in the Playground Grid, instead of distances moved.

SHOW OBJECT is a nice, self-explanatory command.

## ✔DESTROY OBJECT

I know, I know. You worked hard to construct your object, so you're wondering why you have to destroy it now. Here's the reason. Whenever you are running FUNdaMENTAL on your computer, you have a certain amount of the computer's memory to play with. The trick of good programming is to use that memory as efficiently as possible. Each object that you construct takes up a little bit of the

computer's memory while your program is running, and that's okay, as long as you are still getting good use out of that object.

Using this command really involves thinking about how your program runs through *time.* As you start to make more complex programs, you'll find that some objects may have a big role at the beginning of your program, but then aren't used at all later on. This is fourth-dimension stuff that's pretty heavy duty. Still, it's a good idea now to get in the habit of telling the computer to destroy each object once it's no longer needed in your program, even if the program is about to exit anyway.

## What's Going On Inside the Computer?

Those of us without a strong technical background are used to thinking of "the computer" as something like a genie in a plastic box. But of course, when the commands CONSTRUCT OBJECT/PLACE OBJECT/SHOW OBJECT result in a cartoon image of an alien appearing on the Playground, it's not because "some little guy in there read the instructions and just did it." (although it may be tempting to think about it that way).

How is it really accomplished?

FUNdaMENTAL is an assembly language, which means that it sends commands directly to the computer hardware that it controls, specifically, the Central Processing Unit, or the CPU. The CPU itself is divided up into various components. One of the most important of these for our purposes is called the Accumulator, or the AC to its friends (who now officially include you!). The AC is the place where the computer's current "thought" (or data) is contained. For example, the CONSTRUCT command constructs an object from the specified blueprint, and puts that new object into the AC. The MOVE command moves whatever is in the AC (as long as that thing is an object) in whatever way you tell it. Similarly, the LOAD commands that you'll be learning later "load" different things like numbers, sounds, or strings of text into the AC. Other commands then tell the AC what to do with whatever it currently contains.

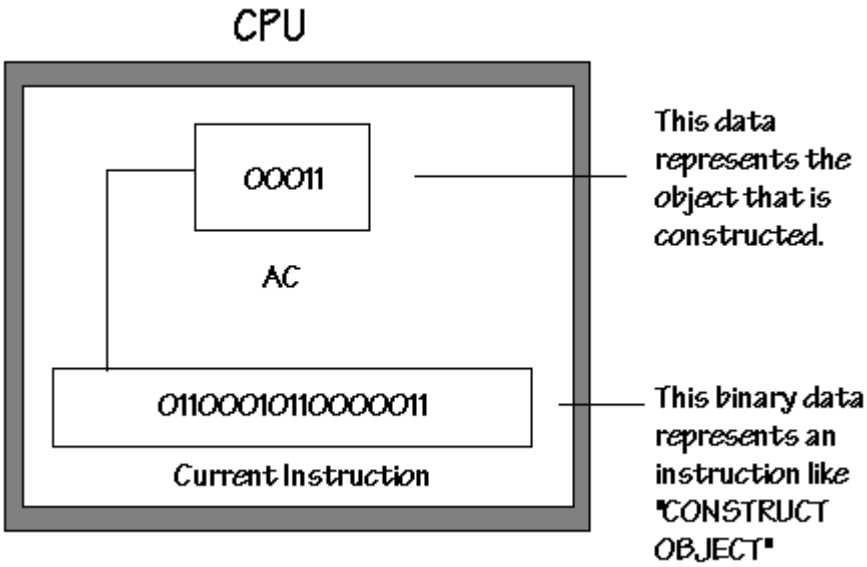
There are two main rules for dealing with the AC:

1. The AC can only contain one thing at a time.

2. The AC can only do things with whatever it's contains at the time. (For example, it can't play a sound after you've told it to construct an object...)

Which means:
Knowing what's in the AC at all times is essential for successful programming!

But wait! Can the computer read English? Not really. The computer interprets everything you tell it as a binary number. Binary numbers are strings of 1s and 0s. Looooong strings of 1s and 0s! (For example, if you were to ask the computer to load the word "hi," here's how the computer understands that word in "binary speak": 0100100001101001–kind of makes you grateful the computer can't talk back, doesn't it?) Which is exactly why we need computer languages like FM to act as official tour guide and interpreter as we navigate the computer's inner landscape.




Illus. 3A The computer understands commands and data as binary numbers

## The Debugger

Once you get used to thinking about how your program actually maps onto the inner landscape of the computer, you'll be much more effective at rooting out bugs in your programs. FUNdaMENTAL has a little feature called the Debugger which can help. The Debugger has a window in the FM interface that helps you find places in your program where you and the computer may not be communicating in the way you intended. You use it when a program doesn't run the way you expected it to, or doesn't run at all! For any given instruction in your program, the Debugger can tell you the current contents of the AC. It can tell you a bunch of other things, too, which will be useful as you learn more about what's going on inside the computer. Here's what the Debugger looks like:

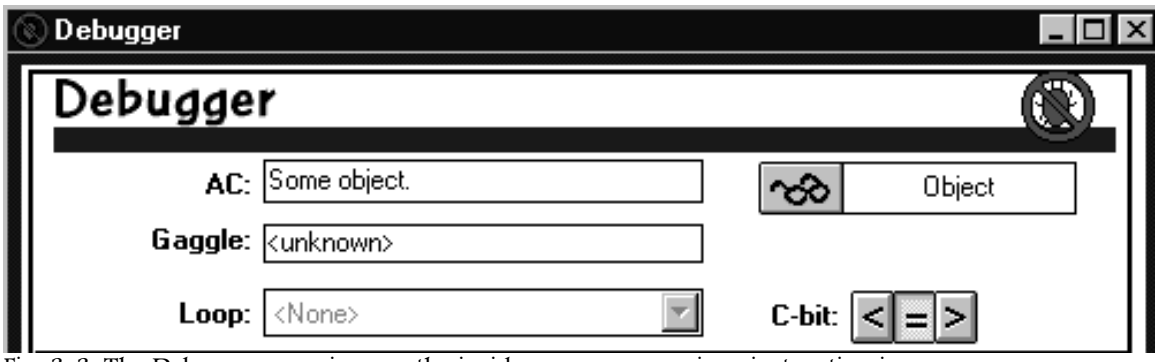


Fig. 3-3 The Debugger can give you the inside scoop on any given instruction in your program

Don't let all the other stuff in this window scare you out of using the AC field and the data field to help you debug your programs. It's a great tool, and it can really strengthen your understanding of how you and the computer are communicating.

Here's how you open it up.

Scan through your program to find an instruction in the list that you think might be causing a problem. (Remember, the Debugger is just a tool for you to use. You're still the one in charge of rooting out the bugs in your program!)

Click on the little round stop button that's in the Instruction list, just to the left of the instruction you want to inspect on the inside. The dot should turn from white to red. This makes a break point, or stopping point, in the code.

Click on the Play button on the Toolbar and then click immediately on the Debugger button on the Toolbar. (You may notice a little blue arrow that appears in the middle of the red dot next to the instruction. This means that the computer will execute this instruction, and then pause.) The Debugger window will then open up. You can drag the Debugger window over to the side and see it simultaneously with the Task window.

The instruction you see highlighted and marked by the little blue arrow is the one that the computer has paused at before executing. In order to get the computer to execute this command, and display the results in the Debugger window, you need to click on the Step button on the Toolbar. It's the little button with a shoe print on it. Doing this will cause the computer to execute the instruction you've highlighted and pause before the next one (which will now appear highlighted and marked with the little blue arrow). By using the Step button repeatedly, you can "step" through the whole task and see your program executed instruction by instruction, with the computer's inner workings mapped out in the Debugger window. (Just remember that you won't see the results of the highlighted instruction in the Debugger window until *after* the Step button has been clicked.)

Visit the Debugger frequently. The more you do, the better your programs will be and the more you'll understand and appreciate the information the Debugger offers.

## LET'S PROGRAM

1. Launch FUNdaMENTAL and open Program3.fmp. At the Welcome window, click the Open Existing Programs button. In the Open Program window, double-click on the following folders: c:\, Fundam, Manual, Practice, Program3. Open that program by double-clicking on Program3.fmp in the File Name list on the left. When you open it up, you should see a task list that is blank, except for the command EXIT PROGRAM. (Yoiks! This is all about you! But don't be fooled by appearances; your program will soon take off and fly!)

2. Use CONSTRUCT OBJECT. Click the cursor in the text-entry field and type or select the command, CONSTRUCT OBJECT. Click the Use button, and choose one of the available object blueprints by clicking on a name in the Data Wizard list.

3. Tell the computer to PLACE the OBJECT. Type or select the command PLACE OBJECT and click the Use button. Type in numbers for the x and y coordinates when the Data Wizard shows you the dialog box .

4. Remind the computer to SHOW the OBJECT...

5. Test run your program. Click the Play button on the Toolbar and applaud when your new object appears in the Playground window!

6. Now use MOVE and MORPH to animate your new OBJECT.

7. Use similar sets of commands to add more objects to the scene.

8. At the end of your program, use the command DESTROY OBJECT for every object in your program so your computer knows it can free up the little bit of memory space it set aside at the start of your program with the CONSTRUCT command.

## Programming Tips and Tricks

To make a really fun animation, you'll want to have a fairly long list of commands. You can use the Control Key-I keyboard command as many times as necessary in order to cause that EXIT PROGRAM command to scoot down and make room.

Construct another object, or two…or three! Notice what happens when you start trying to work with more than one object in your animation programs. Read on to the next chapter for instructions on how to keep everything moving.

But before you go on, did your program run okay? Even if it did, experiment with the Debugger to see a breakdown of how each of your instructions affects the data in the AC, or moves data in and out.

# CHAPTER 4
## Using Boxes to Keep More Than One Frog Hoppin'



Commands Introduced:

## ----- Teacher's Journal -----

I was raring to go, ready to make a veritable plague of frogs in the Playground window. That, for sure, would impress my husband! I told the computer to construct five or six of them and place them all around the Playground. Then I inserted MOVE OBJECT commands for every froggie.

Hop! went the first one, and then sat still. Hop! Hop! went the second and third ones, each in turn. And on they went, one at a time, like popcorn kernels that popped once and then played dead.

"Hmmm," I said to myself. "Not QUITE the effect I had in mind…"

## Moving More Than One Object

If you constructed more than one object in the last practice exercise, you may have noticed something funny (and a little frustrating!). Once you have more than one object on the screen, it's natural to want to make them run, or jump, or fly, or waltz, or wrestle at the same time.

It starts out okay if you construct/place/show one object, and then use some commands to move it around. But as soon as you do it again with another object, the computer acts like a baby who sees a bigger lollipop. It drops the first object and "forgets" about it forever.

UNLESS…! Unless you use commands that tell the computer to store away the memory of the first object before it moves on to the next, and later use commands that tell the computer to recall the stored-away object so it can be used again.

Look at this sample code and see if you can figure out what does what:

```
CONSTRUCT OBJECT froggy
PLACE OBJECT (50, 0)
SHOW OBJECT
STORE BOX frog1box

CONSTRUCT OBJECT froggy
PLACE OBJECT (100, 0)
SHOW OBJECT
STORE BOX frog2box

LOAD BOX frog1box
MORPH OBJECT longfrog
MOVE OBJECT (50, 0)
MORPH OBJECT shortfrog

LOAD BOX frog2box
MORPH OBJECT longfrog
MOVE OBJECT (50, 0)
MORPH OBJECT shortfrog

LOAD BOX frog1box
MORPH OBJECT…
```

If you kept up in this manner, what's your best guess as to how the program would look if you ran it?

## What's Going On Inside the Computer?

Remember the rules of AC? It can only contain one thing at a time, and it can only do stuff with what it contains. If you have something in the AC that you want to put aside now, but use again later, you have to use another part of the computer's hardware to preserve a copy of it.

The STORE BOX command creates a labeled "box" in the computer's memory space, and then fills that space with a copy of whatever's in the AC at the time you use the command. In the sample code above, even though the second CONSTRUCT OBJECT "froggy" boots froggy number one out of the AC, he isn't lost forever. There is now a copy of him "in storage."

The LOAD BOX command copies things back into the AC from the storage space. But it does this indirectly, by calling for a box name, instead of for the object itself.

You can load objects back into the AC, and then move them or even morph them, and the stored copy will be automatically updated according to your changes.

For example, if you load the frog in Frog1box into the AC, and then morph the AC copy into a prince, Frog1box will now contain a copy of the prince as well.

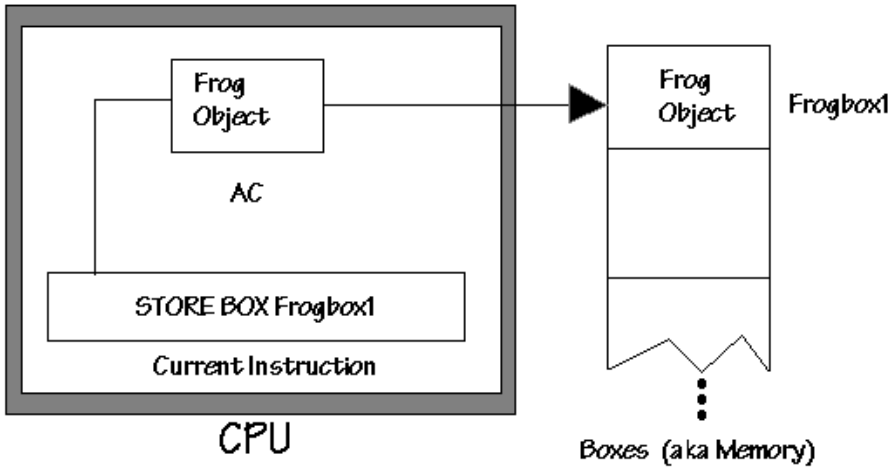



Fig. 4A STORE BOX places a copy of the AC contents in memory

✔STORE BOX
✔LOAD BOX

When you use the STORE BOX command, the Data Wizard reminds you to name your box with some word or words that will later help you remember what's in it.

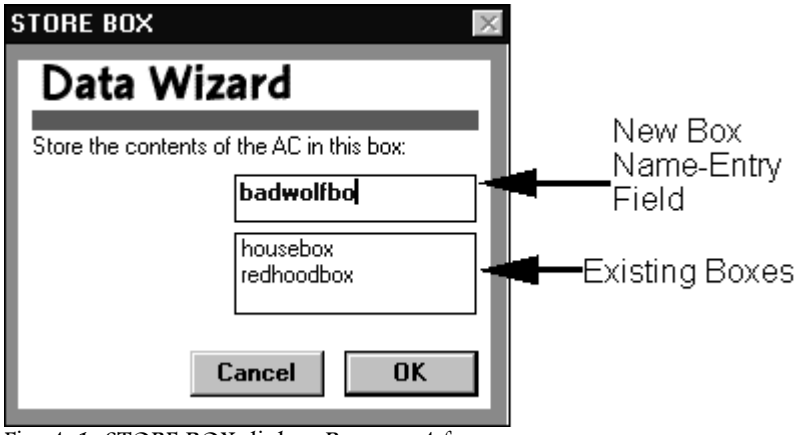



Fig. 4-1 STORE BOX dialog, Program4.fmp

You can just type in an appropriate name for your box and press return when you are done. Now it's time to make your box official.

The Data Wizard will ask you to do so with a little dialog box that looks like this:

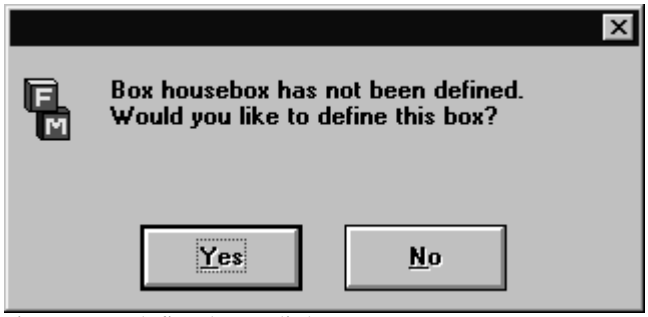



Fig. 4-2 Undefined Box dialog

The computer can't work with a box unless it has been named and entered into an official box list, which is to the right of the Instruction list in the Task window. One thing you should know is that box names cannot contain a space, so if you're using more than one word, run them together like this: "frogbox." Clicking Yes in this dialog will "define" the box by entering it into this list. (If you click No, then you'll return to the dialog and the new box name will not appear in the list. You can either change the name of the new box that you want to define, or cancel the whole thing by clicking on the Cancel button.)

Now that the box is "defined," it will appear in the list of available boxes in the region to the right of the code in the Task window.

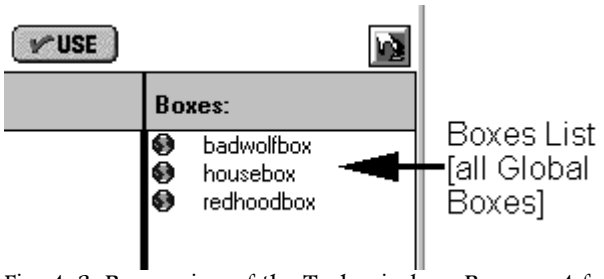



Fig. 4-3 Box region of the Task window, Program4.fmp

Notice that there is a little globe symbol next to the box name in the list. That signifies a "global box," which means that whatever's inside can be called into any part of your program at any time.

When you use the LOAD BOX command, the Data Wizard will ask you to choose from the same list of official, or defined, boxes.

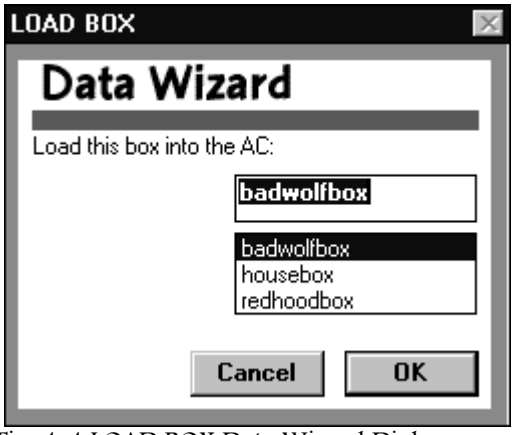



Fig. 4-4 LOAD BOX Data Wizard Dialog

You can do this by clicking on the correct name in the list of boxes. The name should appear in the empty space at the top of the Data Wizard dialog. Click OK when you're sure you've selected the correct box. (At this point, that shouldn't be too tricky, since there's only one!)

## Boxes

The nifty thing about boxes is that they can help you hoard a whole variety of things, from objects to sounds to numbers to text strings to advanced programming stuff like gaggles and booleans!

The rules for boxes vary slightly, depending upon the kinds of data they're storing. But in general they always serve the same handy function in your programs. We'll be seeing a lot of them!

# LET'S PROGRAM!

1. Launch FM, and open Program4.fmp. At the FM Welcome Window, click the button to Open Existing Programs. In the Open Program window, double click on the folder markedFunddam under c:\ in the directories list, and then navigate through Manual and Practice until you find the folder for Program4 and double-click on it. When Program4.fmp appears in the file names list to the left, double-click there, to open the program. The Instruction list should show those old familiar CONSTRUCT/PLACE/SHOW commands for two different objects, Red Riding Hood and the Wolf.

2. Use STORE BOX. Insert the new command after the first SHOW OBJECT command. Click the Use button after you have typed or selected the STORE BOX command.

3. Use the Data Wizard to name the box. Don't forget to use descriptive names so you know what's in them.

4. Repeat steps 2 and 3 for the second object. Look for the second SHOW OBJECT command, and insert your new STORE BOX underneath.

5. Now use LOAD BOX underneath all the existing code, and above the command EXIT PROGRAM. (You may need to insert some lines to make room above the command EXIT PROGRAM. Do this with the keyboard shortcut Control Key-I.). Click the Use button to see the two boxes from which you can choose for loading into the AC.

6. Select the box containing the object you wish to move first. Click Done to enter this instruction into your list.

7. Use MOVE OBJECT. The AC will now move the object that was stored in the box you just loaded.

8. Repeat steps 4, 5, and 6 for the second box, and continue alternating between the boxes with MOVE commands in between. Get Red and the Wolf both moving through the woods toward Grandma's house.

9. Spice up the adventure! Add Grandma and Woodcutter objects, and get them going too. Or go back to the program you created in the last chapter, and use boxes to add more characters and actions.

## Programming Tips and Tricks

Once you have a program with boxes, take a look at what the Debugger can do for you. Refer back to chapter 3 for a reminder about how to open up the Debugger window. Notice that in the bottom portion of the Debugger window, you can see a list of all the boxes used in your task. You'll find yours under the Global Box list. This may not seem so vital now, but it gets more and more useful when you have longer programs with dozens of different kinds of boxes.

If you need help getting started with this exercise, check out the code sequence on page[*insert correct page number for code sequence*] of this chapter. You can use it as a model, just to remind you what command goes where. Once you get going, you can rock the house!

Getting sore fingertips? Read on to Chapter 5 for nifty tricks to save you from typing the same code over and over.

# CHAPTER 5
## Using Loops to Repeat Yourself with Style



Commands Introduced:

## ----- Teacher's Journal -----

The assignment was simple. Use the MOVE OBJECT command to make the little spaceship in the upper left corner of the Playground fly over and land on the head of the stick-figure teacher (a KidPix self-portrait), who stood unsuspectingly at the lower right.

Pairs of students went to work, and within a few minutes we had several versions of the same alien abduction running on monitors around the room. In some, the ship made a straight beeline and crash-landed on the teacher's head. In others, the ship hovered, loopdy-looped, or skulked menacingly below the horizon before swooping down upon the victim.

Everyone was finished, except for Manuel. Despite the urgings of his partner, LaToya ("just LEAVE it!"), Manuel was determined to add some feature to his program. He was leaning forward in his seat, pecking away at the keyboard, and emphatically waving off all the would-be audience members.

Finally, he was ready. We all gathered around, eager for his demo. As we watched, the little ship flew across the screen, landed on the stick-teacher's head, and proceeded to pulse up and down repeatedly for several seconds.

"Brain scans!" Manuel announced, his own face fairly abducted by a huge, satisfied grin.

"Awesome!" we all agreed.

"Hey, Manuel," I said. "Let's see the part of the code that makes it go up and down like that."

Manuel scrolled down his program to find the place where the same pair of Move commands was repeated over and over. It looked like about 50 lines of code. (MOVE OBJECT(0,-10)/MOVE OBJECT(0,10) etc.)

"About how many times does it scan, do you think?" I asked.

Manuel looked at LaToya, who shrugged. "I don't know, maybe 20?" he said.

"All I know is, it took us forever!" LaToya said.

"What do you mean, 'Us'?" Manuel retorted.

"How'd you like to try something like, say, 4,000?" I said, enjoying the sight of Manuel's dropped jaw, as LaToya nearly fell out of her chair. "Let me tell you about a couple of new commands…"

# Do That Again!

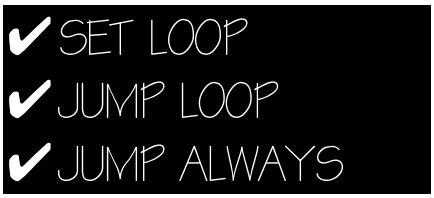
Okay, so we haven't been quite fair. If our experience with these things holds true, at some point during the previous practice exercises you: a) sat there diligently like Manuel, repeating the same sequences of code over and over again in order to keep your program running a little longer, all the while thinking to yourself, "There has to be a better way!" or b) like LaToya, you got disgusted and gave up as soon as you got the point, thinking to yourself, "There has to be a better way!"...a better way, that is, to repeat code sequences without actually having to sit and type them in the Task window, over and over and over.

Well, in either case, you were right. There is a better way, especially now that you know what short work the computer makes of your laborious efforts.

When LaToya and Manuel left class that day, their spaceship was still boinking up and down, somewhere on the way toward completing 4,000 brain scans. Here's how their new code looked:

```
    SET LOOP 4000
@scan
    MOVE OBJECT (0,-10)
    MOVE OBJECT (0, 10)
    JUMP LOOP @scan
```

Before you read on, take a minute to really look at the commands and see if you can tell what's going on.




When you want the computer to repeat a sequence of code, you must first use a command that tells the computer the number of repetitions you want. (Remember, you can't leave anything up to the imagination, because your computer does not have one!!)

When you use the command SET LOOP, the Data Wizard will ask you how many times you want the action repeated.

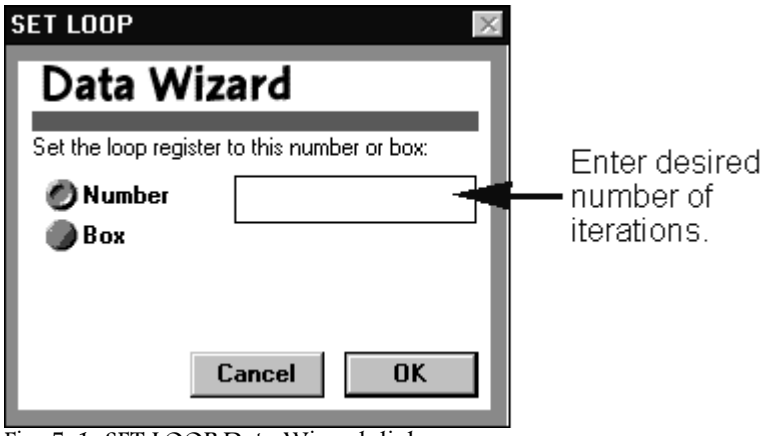


Fig. 5-1 SET LOOP Data Wizard dialog

Notice that just telling the computer SET LOOP 4,000, wasn't enough for Manuel and LaToya to get that little spaceship to actually do 4,000 brain scans. They also needed the command JUMP LOOP, to remind the computer when it was time to "jump" back up and repeat something that came before, instead of continuing on down the list.

## What's Going On Inside the Computer?

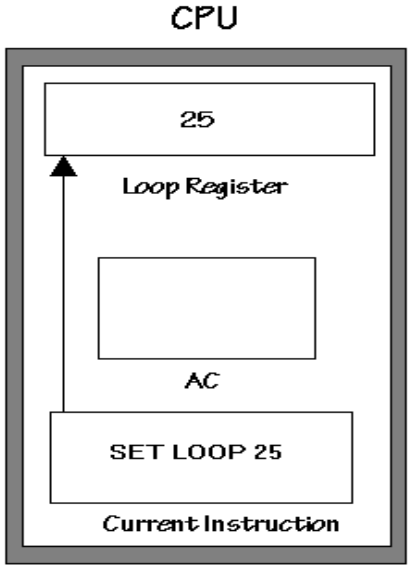
The command SET LOOP brings in another part of the computer called the Loop Register. This is like the little counter they use at the ticket booth to keep track of how many people have gone into the fair. Here, it keeps track of how many times the computer has run through a particular piece of code. Unlike the counter at the fair gates, the Loop Register counts backwards, starting with the number you specify with the SET LOOP instruction, and going down (or in techie speak, "decrementing") one number for each loop until it finally reaches zero. A reading of zero on the Loop Register signals the computer that it can finally move onto the next instruction after the JUMP LOOP command.

JUMP LOOP is in a whole different category of commands from those we've seen so far. It doesn't put a value into any part of the CPU. Instead, it gives directions to the computer about how to follow the other instructions in the list, by way of telling it to jump up to repeat previous instructions, and to keep on jumping until all 4,000 repetitions were checked off in the Loop Register.

(If class hadn't ended before the 4,000 was completed, Manuel and LaToya would have seen how the computer finally finished its task and proceeded to ignore the JUMP LOOP in order to move down to EXIT PROGRAM).

Using these two commands is an example of good programming style because it makes it a lot easier to read and to change the commands in your list. You may be tempted to think of this as more efficient

programming, but that's not actually the case. In programming, efficiency is measured by how many steps the computer has to take in order to complete all of your instructions. Regardless of how fast it goes, the computer will still do every little thing you tell it to. Even though Manuel and LaToya saved themselves a lot of work, the computer still had to go through the same number of steps that it would have done had they chosen to sit there forever and type in 8,000 MOVE OBJECT commands.




Illus. 5A The Loop Register

## Markers

When you use the command JUMP LOOP, the Data Wizard will ask you to identify the place where you want the repetition to begin.

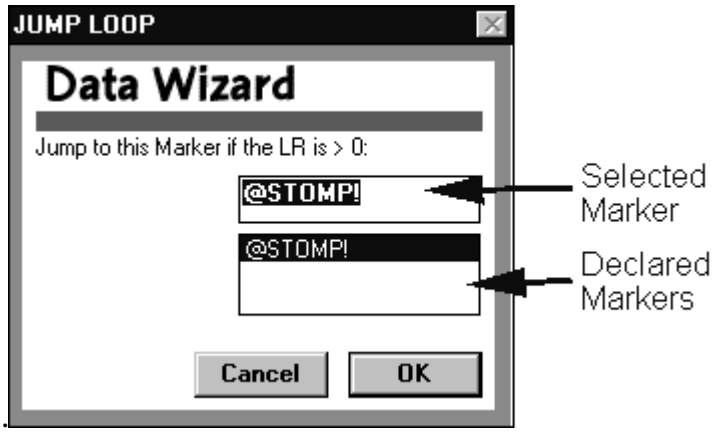



Fig. 5-2 JUMP LOOP Data Wizard Dialog

"What???!" you ask. "I thought I already did that with the whole SET LOOP business…"

Nope. SET LOOP does what the command name implies (in terms of setting the Loop Register) and not a thing more. So every time you use the SET LOOP command, you need to follow it with a marker to mark the beginning of the sequence that needs to be repeated. Manuel and LaToya marked the spot with this marker: @scan.

A marker is not a command. It gives no instructions to the computer in itself. This is reflected in its slightly off-center orientation in the Instruction list. It's also a different color.

## ✔ JUMP ALWAYS

And while we're jumpin' around... another useful JUMP command is JUMP ALWAYS. It is similar to JUMP LOOP in that it also tells the computer to jump to another place in the code instead of just moving down to the next command down on the Instruction list. You need a marker in place before you use this command too.

Can you guess how it differs from JUMP LOOP? (Hint: JUMP ALWAYS works by itself with its marker. It does not require a SET LOOP command.)

# LET'S PROGRAM!

1. Launch FM and open Program5.fmp. Start at the Welcome window. Click Open Existing Programs, and navigate through the directories folders to Program5. (c:\Fundam\Manual\Practice\Program5). Open Program5.fmp, and look at the existing code. You should see commands to construct one Jack object which is getting stomped by one giant's boot object one time.

2. Notice the little note in plain English which says "Jack starts running from the giant here" inserted into the code for this practice program.

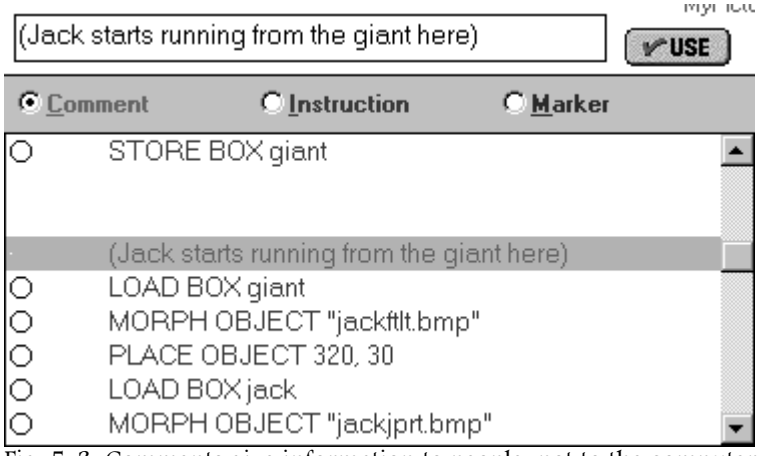


Fig. 5-3 Comments give information to people, not to the computer

That's called a comment. It doesn't tell the computer anything, but it's very useful for humans, as I'm sure you'll agree. You can make comments in your code any time you want, by clicking on the little radio button next to the word "Comment" which appears above the Instructions list. You can add or delete them the same way you do regular Instructions.

3. Examine the instructions responsible for the stomping, and test run the program to see what it does so far. You'll use the Play button on the Toolbar for this.

4. Go back to the Task window and use SET LOOP. You'll highlight the line just above the comment, and then click the cursor in the text-entry field and select SET LOOP. Click the Use button and use the Data Wizard dialog to specify how many repetitions of the action you have in mind.

5. Place a marker underneath the SET LOOP instruction so the computer knows which instructions to repeat. Start by inserting and highlighting a blank line under SET LOOP, and clicking on

the radio button labeled “marker” above the Instruction list. Then enter the text for your marker in the text-entry field, the same way you would type in a command. (Please note that, except for the @ sign, the Instruction Wizard can’t help you with this, since a marker doesn’t exist until you write it!). Click the Use button, or press the Enter key, and the marker will be entered into the Instruction list.

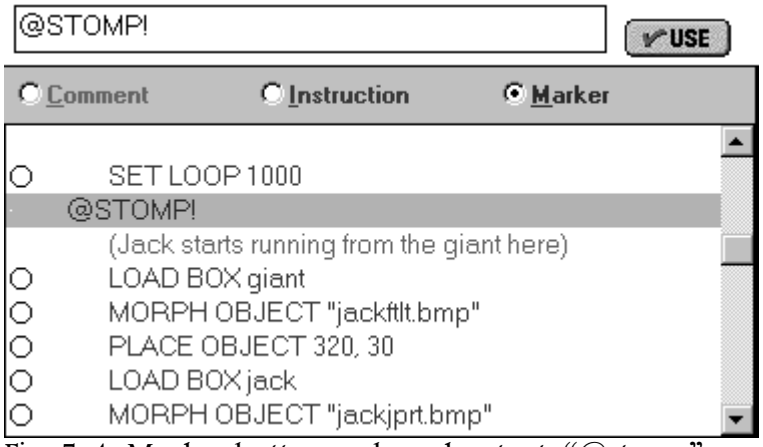


Fig. 5-4 Marker button and marker text, “@stomp”

6. Use JUMP LOOP. Scroll down and find the last instruction before EXIT PROGRAM. Insert a line about EXIT PROGRAM, if necessary, and use either the text-entry field or the All Commands list to select this command. Click Use and then use the Data Wizard to specify the marker that will identify the code which needs repeating.

7. Run your program and see how it looks. Then experiment with different inputs for the SET LOOP command to really get a feel for how it works.

8. Try these commands in other programs. Choose Close Program from the File menu in the Toolbar. From the Welcome window, open up one of the other practice programs you’ve worked on. Play around, framing different sections of code with the SETLOOP/marker/JUMP LOOP commands to get a finite number of repetitions.

9. Experiment with JUMP ALWAYS. Try putting a marker somewhere near the top of your program–say, after an object is constructed, but before you do anything else with it. Then use JUMP ALWAYS as the last command before Exit Program, and specify the marker you just put in near the top. Find a place in between JUMP ALWAYS and its marker and use SET LOOP/JUMP LOOP with a different marker. Your code might look something like this…

```
    CONSTRUCT OBJECT littlefish
    SHOW OBJECT
@repeat
    PLACE OBJECT (0,0)
 @grow!
    MORPH OBJECT bigfish
    MOVE OBJECT (5,5)
    MORPH OBJECT littlefish
    JUMP LOOP @grow!

    JUMP ALWAYS @repeat
```

## Programming Tips and Tricks

Try using loops and markers to move an object a very short distance, say, (1,1),a lot of times in a row. Your code might look something like this:

```
    SET LOOP 300
    LOAD BOX fishbox
 @float
    MOVE OBJECT(1,1)
    MOVE OBJECT(1,-1)
    JUMP LOOP @float
```

Using SET LOOP/JUMP LOOP around sets of MOVE commands with small inputs is a good way to make slower, smoother animation. Aside from looking better, slower animation will be useful later on for programs in which you want the user to be able to click on moving things to achieve a special effect.

Don't forget to check the radio buttons above the Instruction list to be sure that the one you've got highlighted is correct for what you're trying to do!

Once again, the debugger can help you keep track of your loops. If you activate the Debugger and then step through your program, you can see the current Loop Register reading in the Loop field in the Debugger window. For each time the JUMP LOOP instruction is executed, the number in this field will decrease by one. Nifty!

# CHAPTER 6
## A Few Other Goodies



Commands Introduced:

## ----- Teacher's Journal -----

"Hey. You can't see it!" Teodros wailed.

"Can't see what?" I asked.

"These aliens are supposed to land, turn into a bombs, and explode," Teodros said. "You can't see the last part with the bomb."

"Let's see," I said, taking the mouse and clicking on the Play button. A stark landscape appeared in the Playground window, with a lone brick building on the right under siege from a parade of aliens entering and descending from the left. And sure enough, each time an alien appeared, it descended down *behind* the brick building, and the bomb effect was lost.

Across the room, Kulin and Kelley were having a different problem.

"That looks silly!" Kulin was saying.

"I don't think it looks that bad," Kelley said. "I mean after having all that pizza and soda, he SHOULD burp, don't you think?"

"Well, yeah, but that KidPix writing looks silly on there. I wish it could actually MAKE the sound, you know?"

"Yeah," grinned Kelley, beginning to tune up her own repertoire of gaseous sound effects.

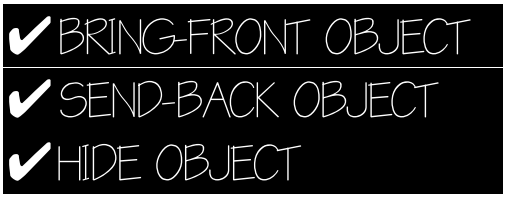
I decided it was a good time to pull the class together and teach them some new commands.

# Sound Cues And Scenery Changes

With the commands you know, you have everything you need to make basic animation come to life. Now you're ready to add the trimmings! How about sound effects? Scenery changes? Extra choreography?

In this chapter, you'll learn eight new commands that really help you spice up your programs. You now know enough about programming with FUNdaMENTAL to begin using these commands right away.

Begin by looking at the list at the head of the chapter. Try to make some predictions about what each command does. Then go ahead and read the explanations below.




These three commands give you more control over how your objects appear, move around each other, and disappear. BRING-FRONT OBJECT tells the computer to superimpose the object that's in the AC in front of the other objects in the program. SEND- SEND-BACK OBJECT does the opposite. HIDE-OBJECTtells the computer to do exactly that. (Great for disappearing acts!)


These two commands give you more control over the Playground environment. If you want your Playground to appear bigger or smaller, use the RESIZE PLAYGROUND command which allows you to plug in the dimensions of the Playground window before you start your program (or anytime in the middle, for that matter).

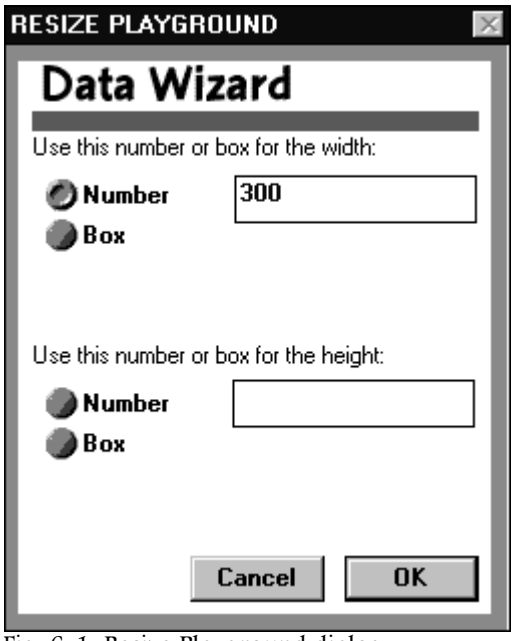



Fig. 6-1 Resize Playground dialog

INSTALL BACKGROUND tells the computer to put up or change the scenery behind the action. When you use this command, the Data Wizard will offer you a list of all the available graphics in the Graphics Library.

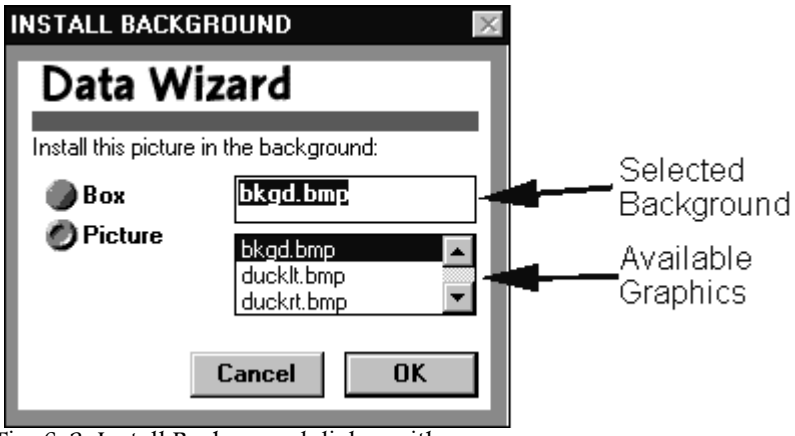
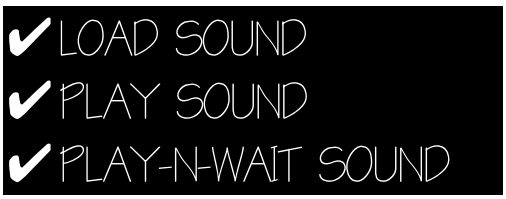



Fig. 6-2 Install Background dialog with available graphics from program3fmp

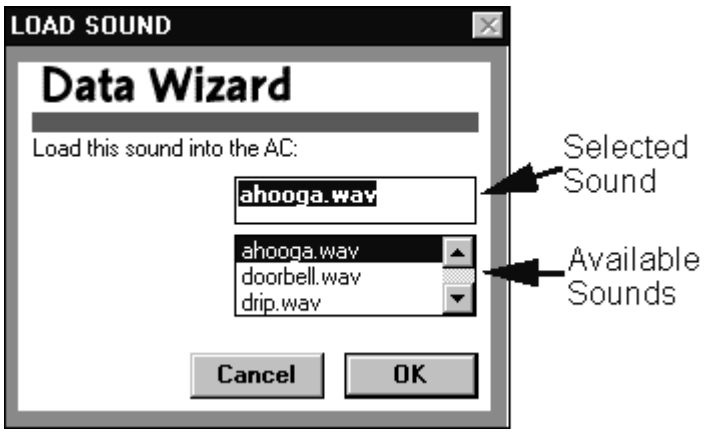
Any graphic can serve as your background, from Green Forest to Green Frog. Experiment and see what happens when you tell the computer to install different graphics as the background for your program. Again, this is a good command to use at the beginning of your program, but you can use it to get a change of scenery any time.




(Hey! It's a new kind of data! Data is the techie term for all the kinds of stuff you can manipulate in your program. Remember, objects are only one kind of data that you can manipulate with FUNdaMENTAL. There are five other kinds. Two of the data types have funny-sounding names: gaggles and booleans. The other data types are more self-explanatory: numbers, text -"strings," and sounds.)

You can use the sound commands to add sound effects to your programs. Any sound is possible; spoken messages and instructions, soft music, or animal sounds can be at your service. (Rude noises of all kinds are always a favorite among my younger students.)

When you use the command LOAD SOUND, the Data Wizard will show you a list of sounds in the Sound Room which are available for you to use.




]

Fig. 6-3 LOAD SOUND dialog with available sounds from Program3.fmp

(You may have chosen to take a look at the Sound Room in the previous unit. You can do so from anywhere inside FM by pulling down the Window menu on your Toolbar, and double-clicking on Sound Room. You can also get there by clicking the headphones icon on the bottom of the Program window.)

## What's Going On Inside the Computer?

Once you load a sound into the AC, whatever else that was there before is now gone. You cannot, for example, tell the computer:

```
LOAD SOUND "getalongcow"
PLAY SOUND
MOVE OBJECT (5,0)
```

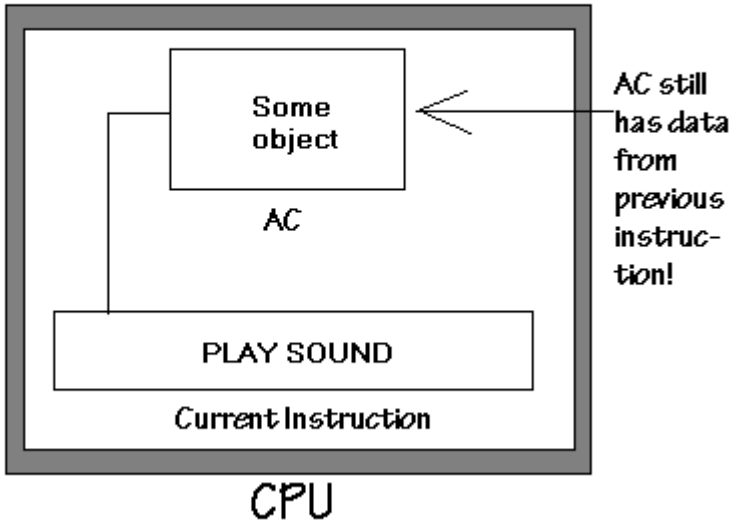
or

```
CONSTRUCT OBJECT cow
PLACE OBJECT (100,100)
SHOW OBJECT
PLAY SOUNDmoo!
```

I know these sequences may seem perfectly logical to you, but the computer isn't so smart. Go ahead and try this sometime, and see what happens. (BONK!) Remember the rules of the AC:

*It can only hold one thing at a time, and it can only do things with what it's holding.*

If you use CONSTRUCT OBJECT to put an object into the AC, and then immediately tell the computer to play an appropriate sound, the computer will essentially answer, "What sound?" Just because you remember that the perfect sound is ready and waiting in the Sound Room doesn't mean that your computer remembers. You have to LOAD SOUND first!




Illus. 6.A Wrong AC type!

## Two Ways to Play Sounds

You may have noticed that there are two different options for playing a sound that has been loaded into the AC: PLAY SOUND and PLAY-N-WAIT-SOUND. Their names give a hint about the different functions they'll serve when you use them in your programs. When you do the next Let's Program section, play around, and see what else you can discover….

## LET'S PROGRAM!

1. Launch FM and open Program3.fmp. We're going back to this program for this exercise. From the Welcome window, click Open Existing Programs, and navigate through c:\, Fundam, Manual and Practice to Program3. Open Program3.fmp.

2. Go to the Sound Room to see what sounds you have so you can make decisions about adding sound effects. Use Window menu on the Toolbar, and select Sound Room. The Sound Room window looks like this:

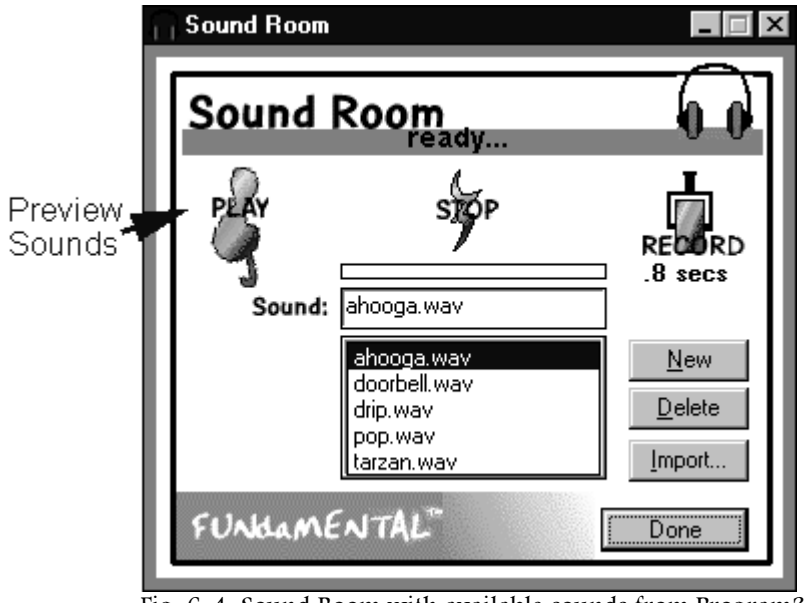



Fig. 6-4 Sound Room with available sounds from Program3.fmp

3. Preview the available sounds. Click on each sound name in the list to highlight it, and then click the Play icon to get a preview of what they all sound like. Click Done when you are finished.

4. Return to the Task window and review the code you wrote for this program. Think about how you might incorporate the

available sounds into your program, and when you'd want them to be played.

5. Add the commands LOAD SOUND/PLAY SOUND or PLAY-N-WAIT SOUND to your program. (You may need to insert some lines to fit these commands in.) Use the list in the Data Wizard to specify the sounds you have in mind. (Note: If you want all the action in your program to stop until the sound has finished playing, then use PLAY-N-WAIT SOUND. If you want the sound to play as the action continues, then try PLAY SOUND.)

6. Test-run the program. Click on the Play button on the Toolbar to see (and hear!) your program run.

7. Now think about adding scenery with INSTALL BACKGROUND. (This is a good thing to do at the beginning of your program code, although you can add or change scenery at any time in your program.) Insert a space, if necessary, and use the command INSTALL BACKGROUND. When the Data Wizard shows you a list of available graphics, choose a graphic that you think would work well here, and select it as the input for this instruction.

8. Do another test-run. Click the Play button on the Toolbar to see how it looks.

## PROGRAMMING TIPS AND TRICKS

It's fun to experiment with different graphics – even unlikely ones!- for your background. Take your time and play around with this. You can get some neat effects.

Try to use all of the new commands introduced in this section at least once. Add background scenery, choreography, disappearing acts, and sound effects to this or other programs that you've created. Dazzle yourself… your cat…your neighbors… (Come on! By now you're ready for a tougher audience!)

# CHAPTER 7
## Be Your Own Graphics Department and Sound Crew

## ----- Teacher's Journal -----

Jaime was a tough nut from day one. He wore baggy slacks and a little ponytail at the nape of his neck, and generally came to class wrapped head to foot in a bad attitude.

The only things about him as noticeable as his attitude were his innate intelligence and his artistic talent.

As soon as we started, he was dying to get his hands on the graphics. All the prefab stamps and stick drawings I used in the practice programs just served to put extra curl in his lip.

"That's LAME!" he would regularly pronounce. "That's SORRY!"

Imagine my delight on the day I was finally able to say, "Okay, Jaime! I'm through torturing you with my lame graphics. From now on, you run the whole show!"

He shot me a suspicious glance, but soon his slouching body was transformed with artistic concentration. With amazing skill, he was using a popular children's draw -and- paint program to create a detailed boxing ring, complete with full-color Mexican and U.S. flags hanging from the ceiling.

I reminded him that the stick-figure boxers had to be drawn on a separate "sheet" since they would need to move around as objects, independent from the background.

"Hey, wrap it up!," I finally said. "That's enough, Leonardo da Vinci. Let's transfer your stuff over to FUNdaMENTAL and start programming."

At first he was reluctant, afraid that pasting his work into the FM environment would somehow "mess it up."

But once he saw his *own* artwork appear in the program Graphics Library, his face cleared of its habitual scowl and filled with interest. He quickly used the Object Designer to create object blueprints using his stick figures. Then he began to program.

He used INSTALL BACKGROUND, and there was his own boxing ring! Then he used CONSTRUCT OBJECT and constructed two boxers according to his blueprints in the Object Designer.

And as his hand-drawn boxers began duking it out in front of their custom-designed boxing ring, Jaime forgot himself completely. He spread out his arms, almost as wide as the smile spread across his face, and announced to the world, "Hey! I'm famous!"

# If You Want It Done Right...

If you are of the artistic bent, or even if you just feel that getting something done right means doing it yourself…then maybe you, like Jaime, have been waiting to find out how to personalize your Graphics Library and Sound Room with homemade (or at least hand-picked) supplies. Perhaps you've asked yourself when you would get a chance to design your own object blueprints in the Object Designer. If you haven't asked yourself these questions, it's time you did! And if you have, then the time is now!

Up until now, we've provided you with ready-made goodies from the Graphics Library, the Object Designer, and the Sound Room so you could concentrate on how to manipulate different kinds of data with commands. But now that you are an experienced programmer (you are!), you're ready to supply personalized, hand-picked data for your programs.

You can always add to the Graphics Library and Sound Room of existing programs. But once you're ready to bring in your own stuff, you're ready to start from scratch with a whole new program. So let's begin with that.

## LET'S START A NEW PROGRAM!

1. Create a new FM program. At the Welcome window, click on the Create New Program button. (Congratulations! This is a first!) You should see a window that looks like this:

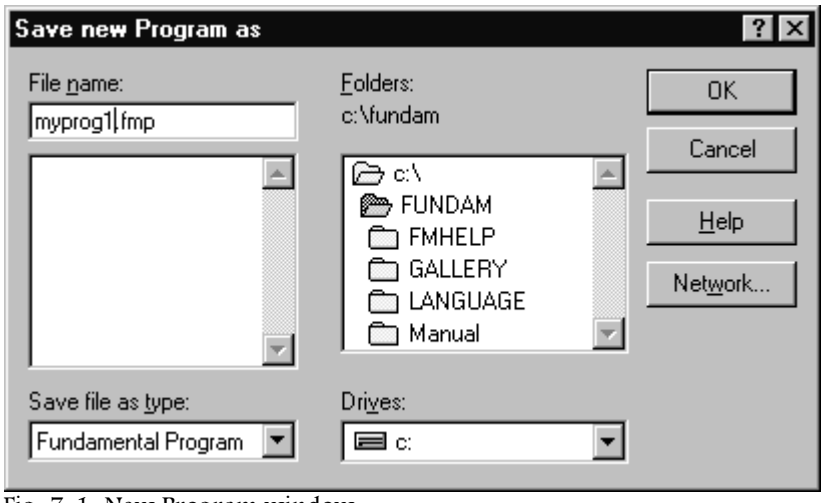



Fig. 7-1 New Program window

This is where you name your program and make a new folder for it.

2. Name the program. Click the cursor between the asterisk and the dot in the new file name *.fmp which appears in the text-entry field. Hit the backspace key on your keyboard once to get rid of the askterisk, and type the name myprog1 in its place to the

left of the dot. (This is short for My first program.) The whole thing should look like this: Myprog1.fmp.

3. Save the new program file in the fundam folder. In the Directories list, double-click on the folder marked fundam to open it. The text under the word "Directories" (which describes the path you'll follow to get at this new file once it's saved), should read, c:\fundam.

4. Click OK, and you'll automatically end up at the Task window for your new program. Please note: if you quit out of this program and want to work on it again later, you'll find it with the other existing programs.

When you choose to make a new program from the Welcome window, you'll not only end up with an empty Instruction list, but you'll have an empty Graphics Library, Object Designer, and Sound Room as well. You'll need to stock up before you can get started programming.

## Personalizing Your Graphics Library

In the Graphics Library, you can paste in your own hand-made graphics and import graphics from other libraries and programs within FM.

Bringing in your own artwork is the most fun. You'll get a chance to see your own images come alive in your programs, and feel truly famous, just like Jaime! Here's how to do it:

# LET'S BRING IN OUR OWN ART!

1. Use your favorite draw -and- paint program to make a simple masterpiece. (If your computer has enough memory, you can even keep your FM program open while you do this.) You can also use available stamps if you're not feeling particularly artistic.

2. Copy your picture to the clipboard. When you finish with your design in your draw-and-paint program of choice, use available selection tools to select it, and then use the Edit menu at the top of your screen to copy the image onto the computer's "central" clipboard.

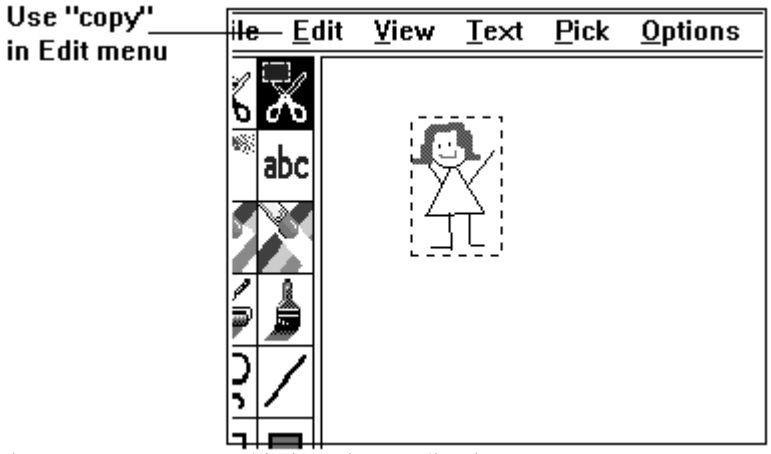



Fig. 7-2 Custom graphic in paint application, ready for export

3. Go back to Myprog1.fmp. If you've quit out of FM, launch it again, and select Open Existing Programs at the Welcome window. In the Open Program window, navigate through the directory folders to find Myprogr1.fmp which you've just created. Open the program.

4. Go to the Graphics Library. Select it from the Window menu on the Toolbar, or get there by clicking the Graphics Library icon at the bottom of the Program window.

5. Define and name a new graphic. In the Graphic Library, click on the New button. A generic graphic name will appear in the graphics list; it should look like this: "image_1.bmp"

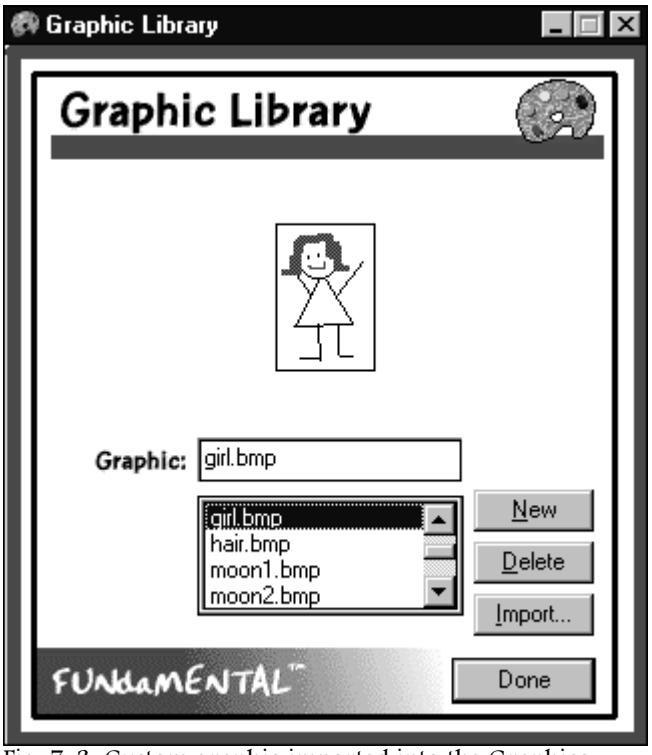



Fig. 7-3 Custom graphic imported into the Graphics Library for Myprog1.fmp

6. You can double-click in the text box to highlight the generic name, and then enter a new graphic name that describes the picture you have waiting on the clipboard.

7. Paste your picture into the Graphics Library. Select "Paste" from the Edit menu…Hey! There's your picture! (fun, huh?)

8. Another option is to import graphics from another FM program (which is most useful if you used your own really cool artwork in one program, and you'd like to use it again in another), or from the FUNdaMENTAL central Graphics Library. Here's how:

# LET'S IMPORT GRAPHICS - From Other Places in FUNdaMENTAL!

1. Go to the Graphics Library and select the Import option by clicking on the "Import" button. That will bring you to the Graphic Importer.

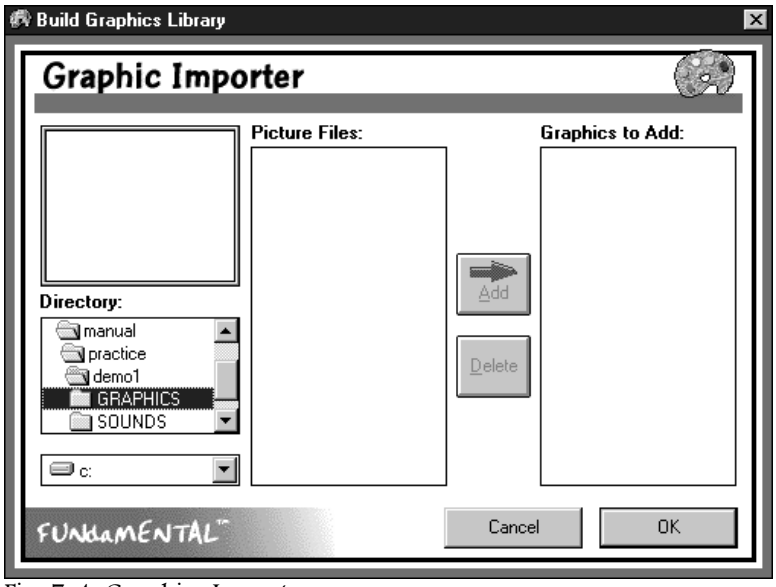



Fig. 7-4 Graphics Importer

2. Navigate through the directories to select the program or library which has the graphics you want. Notice on the left of the Graphics Importer there is a box marked Directory. This works just the same way as the directories box in the Open Program window. Start by double-clicking on the folder marked Fundam to make sure you can see all the choices. Then scroll down the list until you find the program from which you want to grab graphics. Double -click on the directory folder for this program.

3. Get into the graphics. Double-click on the folder marked graphics which will appear among the files and folders beneath open program directory. A list of all the picture files in this folder will be displayed in the Picture Files list in the middle of the Graphic Importer.

   (Note that you may also grab files off a floppy disk or CD-ROM using the drive box, which is directly below the Directory box.)

4. Preview the picture files. Once you're into a stock of pictures, click on any file in the Picture Files box and you will get a preview of the picture in the frame above the directories.

5. Import the ones you want. To import a graphic from this list to your Graphics Library, click on it's name. Then click on the Add

button (it has an arrow pointing right.) The graphic name should appear in the Graphics to Add box.

If you somehow end up with a graphic in your import column that you don't really want, select that graphic in the right-hand list and click on the Delete button.

6. Close the Graphics Importer. After you've finished adding all the graphics you want, click on the OK button. This will bring you back to the Graphics Library, where you can rename your newly imported graphics by following the procedure you learned earlier in this chapter. Click Done when you are finished to return to the Task window. (Make sure you have *at least two* different graphics in the Graphics Library to use in Myprog1.fmp.)

## LET'S TAKE OVER THE SOUND ROOM!

"Oink!" "Vroom!" "Splash!"

In the same way that you created or selected your own images to use in your program, you can record or select your own sounds to use in your programs.

1. Get into the Sound Room. You can do this by selecting Sound Room from the Window menu on the Toolbar.

2. To create a new sound, click on the New button. A new sound name will appear in the sounds list. It should look like this: sound_1.wav.

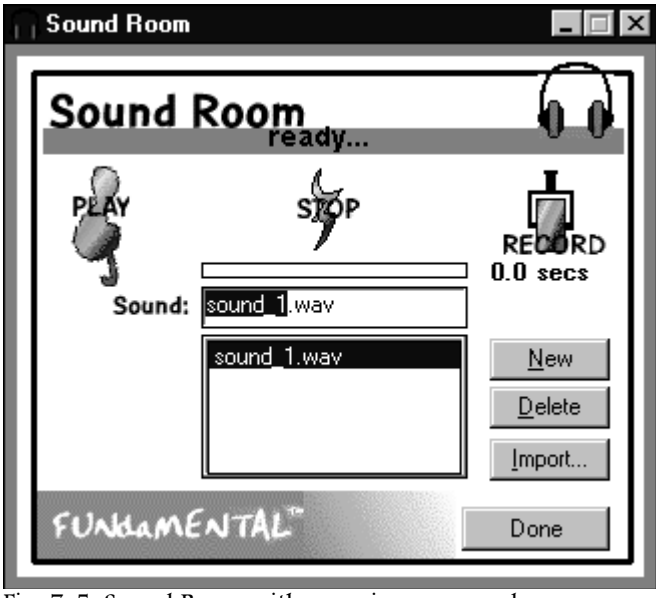



Fig. 7-5 Sound Room with generic new sound name

3. You can double-click in the text box and enter a new sound name. Press return to make it official.

4. To record a new sound click on the Record icon. (You need a microphone of course; many standard tape-deck mics will do fine).

5. To play a sound, click on its name in the sounds list and then click on the Play icon.

6. To delete a sound, first click on the sound in the sounds list and then click on the Delete button.

## LET'S IMPORT SOUNDS
### - From Elsewhere in FUNdaMENTAL

1. Click the Import button in the Sound Room Window.

2. Navigate through the directories to find the program that has the sounds you want. When the Sound Importer comes up, it will look like this:

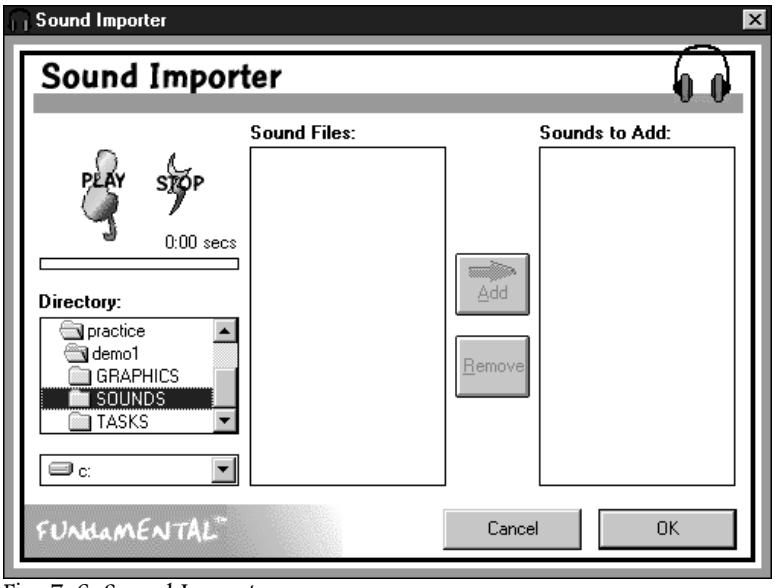


Fig. 7-6 Sound Importer

3. Select the program you want by navigating through the Directory box. When you reach a folder that contains sound files (their name will end with ".wav"), they will appear in the sound Files box. (Note that you may also grab files off a floppy disk or CD-ROM using the drive box, which is directly below the Directory box.)

4. To preview the sound, click on the file in the sound Files box and then click on the Play icon.

5. To import a sound from this list to your Sound Room, click on the name of the one you want. Then click on the Add button (it

has an arrow pointing right). The sound should appear in the Sounds to Add box.

If you somehow end up with a sound in your "sounds to Add:" column that you don't really want, select that sound in the right-hand list and click on the Remove button.

6. Close the Sound Importer. Click on the OK button when you are finished selecting sounds to import, and you will be returned to the Sound Room. Your imported sound names should appear in your Sound Files list.

## Putting Your Pictures and Sounds to Work

Well, now you have a new program with a stocked Graphics Library and Sound Room. What now?

You can make a program that just plays different sounds, and if you're creative you can come up with something neat. But you can really strut your sonic stuff if you incorporate sounds as sound effects for animation.

And we know that FM can't do much at all with those nifty graphics unless you attach them to objects. So now it's time for you to learn how to build your own object blueprints, complete with hand-picked graphics and a few other cute tricks. Read on to chapter 8 to learn how!

# CHAPTER 8
## Designing Blueprints for "Clickable" Objects



New Commands:

## ----- Teacher's Journal -----

Sarah was the first one done with the day's assignment, and she was pretty pleased.

"Come see!" she called, waving me over with an enthusiasm that almost tipped her out of her seat.

I came over and she clicked on the Play button to demo her program. Two identical objects began floating around on the screen. She'd brought in a ghost stamp from a draw/paint program to make them.

"Click on them! Click on them!" she said. I clicked.

The little ghost stopped mid-float and flashed through about 10 ghoulish transformations before finally returning to his original form and continuing his haunt. The other one worked just the same way.

"Pretty darn scary!" I complimented.

"Yup!" agreed Sarah. "Hey, Yolanda! Come look at this...!"

## Object Instances and Object Designs

At the start of her program, Sarah had used the command CONSTRUCT OBJECT spook twice. What she was actually doing was making two instances, or editions, of an object type according to a specific blueprint. The Object Designer allows you to design your own object blueprints.

It's easy to think of an object primarily in terms of how it looks, but there's more to it than that. An object design consists of the following:

- a starting graphic (Once you have an object you can always change its appearance by morphing it...Sarah's starting graphic was called "ghost", but the object was called "spook" because it did a lot more than just wear a ghost graphic.)
- a starting location, in terms of the coordinate grid
- a click-task, which means that any object you construct from this blueprint will do a particular trick when you click on it...like morph into a centipede, or burp loudly.

The code to make this special effect happen could be as short as one or two lines: MORPH OBJECT centipede, or LOAD SOUND buuurp/PLAY-N-WAIT SOUND. (Sarah simply used the MORPH OBJECT command over and over until she had run through all the graphics that she'd hand-picked for her library.)

But these lines can't be in the main list because you only want the computer to follow these instructions *if* the user clicks on the object.

You need a separate place to stash these commands...That's why an object click -task is actually a separate mini-program, or sub-task...

## Sub-task? What's A Sub-task???

(Remember in the introduction, when I said that most programs consist of a bunch of smaller programs called sub-tasks, that are all linked to the main task and sometimes to each other? Well that's okay...I can say it again now.)

Most programs consist of a bunch of smaller programs that are all linked to the main program, and sometimes to each other. (Oh, REALLY???)

An animated program with one or more objects that you can "click on," to make burps or centipedes or whatever, is *one* example of how programs can incorporate sub-tasks.

Since it's really fun to design objects with click-tasks, and click-tasks are really sub-tasks, then this is a good time for you to learn how to make new sub-tasks when you need them.

# LET'S MAKE A NEW SUB-TASK!

1. Open Myprog1.fmp which you created in the last chapter. You'll find it with the other existing programs in the Open Program Directories window: (c:\Fundam\Myprog1.fmp).

   This program will have a blank task list since all you've done so far is gather supplies.

2. Stop and take a moment to envision this: At the end of this chapter, I want you to have a program that shows an object moving around the Playground, and I want it to morph into something else when you click on it. Right now we're going to write the part of the program that's in charge of that morphing click task. It's okay that all the commands for the main task are not yet written.

3. Find the Program window. You can use the Toolbar, pull down the Window menu, and select Program window to do this, or just click on any part of it that may be visible to you.

   The Program window is the directory for everything that's going on in your program. The big white space in the middle shows all the Tasks (or instructions lists) that your program has. Unless you've been a real prodigy and figured out how to add sub-tasks on your own, there should be only one task showing: Main. We're about to change that.

4. Create a new sub-task. Click on the New button near the top of the Program window. The computer will show you a dialog labeled, Create a New Task which asks you to name the new File, but what it really means is you should name your sub-task.

   You might have noticed by now that choosing names in FM is almost as important as choosing names for your pets…If you name your dog, your cat, your iguana, and your canary all Mike, when you yell out "Mike! Dinner!" you're likely to be trampled in a stampede… sub-tasks, like boxes and pets, need distinctive names so you, and the computer, can be certain of what you're calling.)

5. Name the sub-task Type the name of your sub-task into the little file-entry field that's highlighted in blue. Call the sub-task "Morph."

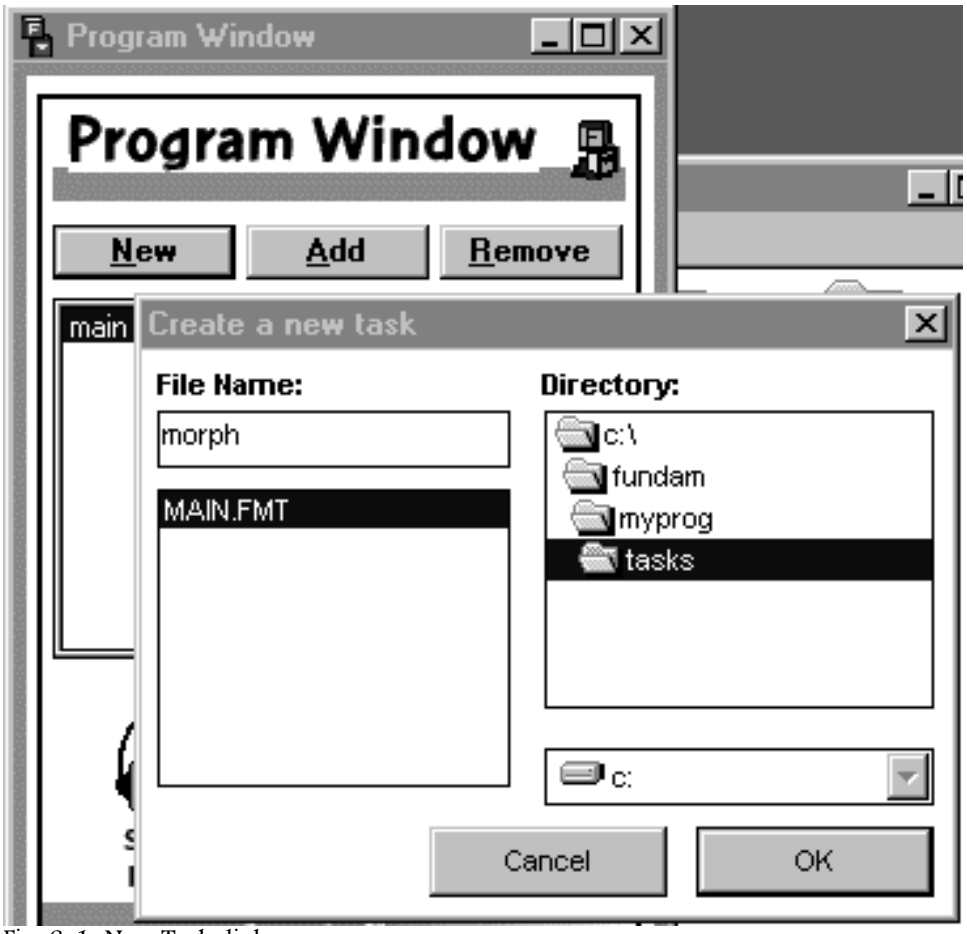


Fig. 8-1 New Task dialog

6. Click the OK button in the Create a New Task Window. You should now see a Task window with a blank Instruction list that looks just like the regular main Task window, except that it has your carefully chosen sub-task name at the top.

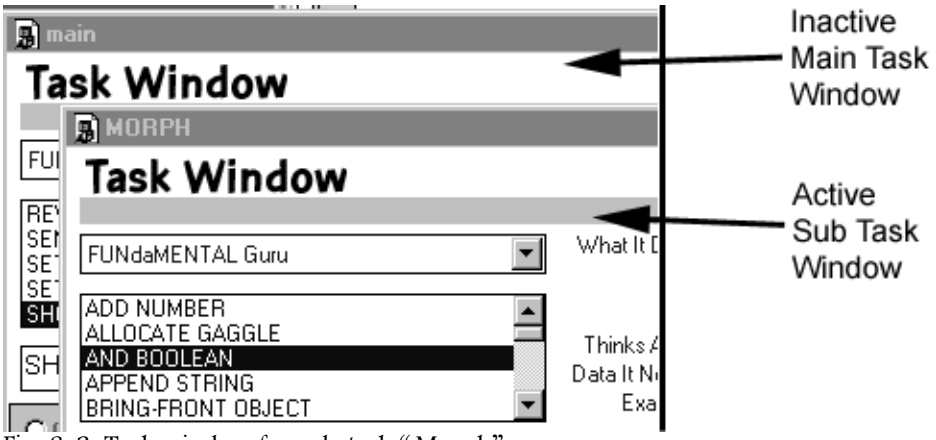


Fig. 8-2 Task window for sub-task "Morph"

7. In the Instruction list for this sub-task, use MORPH OBJECT. Enter the command the same way you have all along. Click the Use button and use the Data Wizard to choose a hand-picked graphic object to morph into…You can add any number of other

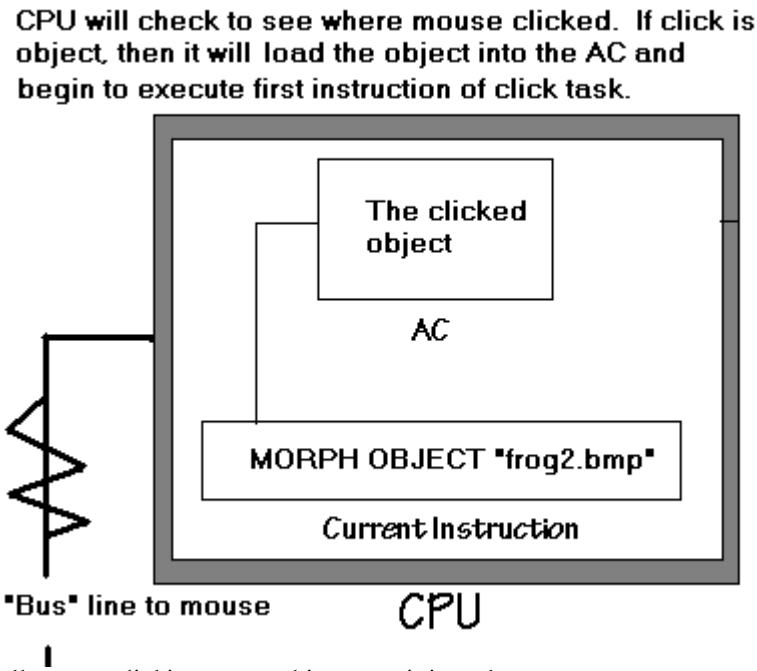
commands to this sub-task at any time, but let's keep it simple for now. Just one command is enough to make a cool special effect for whoever clicks on your object when you demo your completed program.

## What's Going On Inside the Computer?

I know what you're thinking: "What if I have more than one object in my program? If the only instruction in my sub-task is MORPH OBJECT, how will the computer know which of my objects I want it to morph when the mouse clicks?"

Good question. Here's the answer. FM has a little built-in sleight of hand to make click-tasks easy to incorporate into your programs.

The act of clicking on an object automatically puts that object in the AC, and executes whatever commands are in the associated sub-task. So while you would never be able to start your main task with the command MORPH OBJECT, you *can* begin a click-task this way.




Illus. 8A Clicking on an object puts it into the AC.

## Everything You Need...

Now you have everything you need to make a blueprint for an object to construct and use in your program. You have the graphics to give it its look, and a click-task waiting to spice up its act. Time to put it all together in an object design.

## LET'S DESIGN OUR OWN OBJECTS!

To create a new object design, you must first have at least one graphic in the Graphics Library. (If you followed the instructions in the previous chapter, you should have at least two custom-made ones there.)

Once you're in the Object Designer, you'll need to do five things:

- let the computer know you want to design a new object,
- name the new object,
- choose its initial graphic,
- choose its initial location, and
- assign a sub-task, if you have one (which in this case you do!) to be the click-task for your object plan. (Note that you can actually have two different sub-tasks linked to one object: one that executes when the user clicks on the object, and another that executes when the user pushes the Control Key on the keyboard while simultaneously clicking on the object. This second kind of click-task is called a Control-click-task.)

1. Get into the Object Designer. You can do this by using the Window menu on the Toolbar and selecting the Object Designer from there.

2. Click the New button. A new object design will be created that is currently named object_#1.

3. Rename the object. Since object_1 is not a terribly descriptive name, you should name your new object something else. First, double-click on the name object_1 where it appears in the file name-entry field labeled Object. Then rename the object.

4. Choose an initial graphic. Notice that the graphic that currently appears to be associated with the object is the first one alphabetically in your Graphics Library collection.

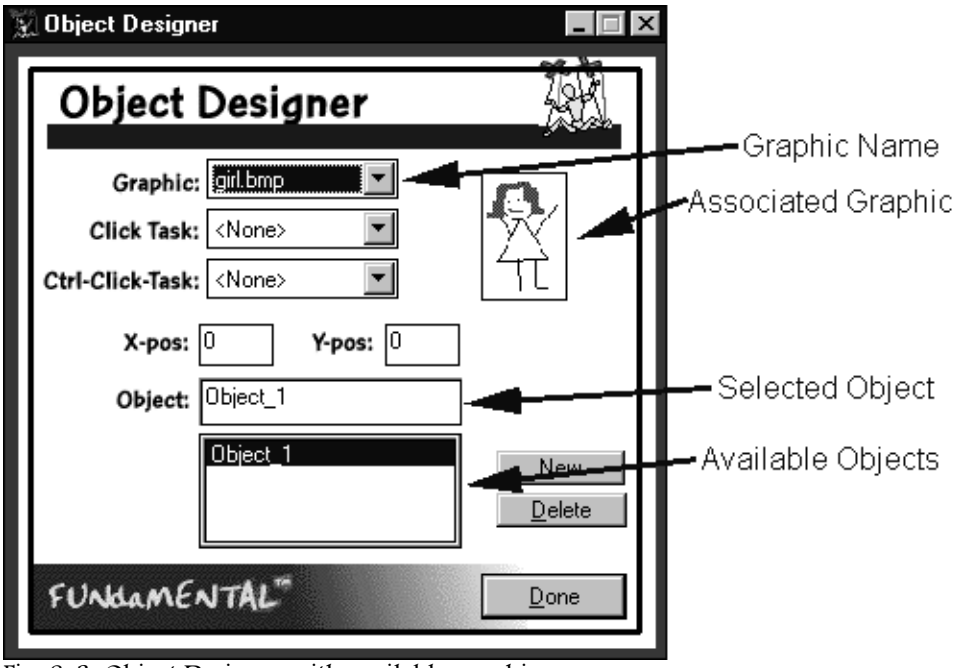



Fig. 8-3 Object Designer with available graphics

If the graphic you see is the one you want as the starting graphic for this object, skip down to step 5.

If the picture you see is not the one you want for this object, then click on the down arrow next to the drop-down menu labeled Graphic, and scroll until you see the name of the graphic you want. When you find it, click on the name so it will appear in the Graphic box, and you can see a little preview of the picture, so you can be sure it's the right one.

5. Leave the location fields at (0,0). Sometimes you may want to write a program with stationary objects, in which case it makes sense to plug location coordinates into your object design. But since this program will have moving objects, you can take care of lacing your object with a PLACE OBJECT command in the code.

6. Assign the sub-task, "morph" as the click-task for this object. Use the drop-down menu labeled Click-Task, and find the name of your new sub-task, "Morph," in the list. Select it by clicking on the task name.

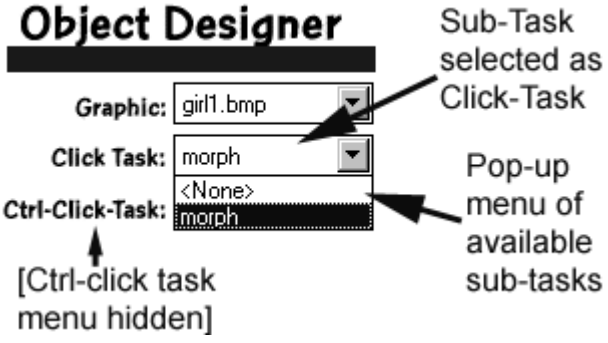


Fig. 8-4 click-task pop-up menu in Object Designer

Now, every object instance of this type will be "clickable," and able to do the morph trick. Leave the Control click-task with "None" for now, but once you get the hang of this, you can link another sub-task to your object here in exactly the same way as you did with the regular click-task.

7. Close the Object Designer. Click on the Done button when you're finished designing your object.

## LET'S PROGRAM!

WHAT ARE YOU WAITING FOR? Now you're ready for your moment of fame…write a program using your hand-picked/drawn graphics, your custom-designed objects, and your original and hand-picked sounds.

1. Get back to the main Task window that belongs to Myprog1.fmp. You can find it by using the Window menu on the Toolbar, and selecting Program window from the drop-down list. Once in the Program window, double-click on the main item in the list of program tasks.


Fig. 8-5 Program window for Myprog1.fmp, with two tasks in the list

Okay, so that was the long way, but I wanted you to see that your Program window Task list now holds two tasks!)

2. Use the command CONSTRUCT OBJECT. Click Use, and the Data Wizard will show you a list containing the object(s) that you designed earlier in this chapter. Select from among them, and click Done to enter the instruction into the list.

3. Use PLACE and SHOW commands to make your object visible. It's a good idea to STORE it in a box, too.

4. If you wish, make another instance of the same object.

5. Use MOVE OBJECT to get things moving. Remember to use SET LOOP/JUMP LOOP or JUMP ALWAYS to get the computer to repeat small movements like (3,3) over and over. This will allow your users a chance to try out the click-task you've included in your object design.

6. Use LOAD SOUND/PLAY SOUND to add sound effects to your program using your hand-picked and custom-made sounds from the Sound Room.

7. Refer all calls from the press to your agent.

## Programming Tips and Tricks

This is where that trick of looping through a lot of slow movements will really benefit your program. You can't click on an object if you can't catch up to it, and the computer is so quick to obey your commands that regular MOVE instructions will likely leave your user out of breath, and your object unclicked.

When you add your sound effects, don't forget the flexibility you have with the two different PLAY commands. One flicks the Play switch, and then goes about its business while the sound plays. The other stops everything to wait until the sound has finished playing.

## How Quick Is Your Clicker Finger?

Having trouble beating the computer to the punch (or should I say "to the click")? Has it zipped through the commands and gotten over the Exit Program finish line before you've even got your hand on the mouse? There has to be a way around this. After all, it's great to have such speedy delivery on your instructions, but sometimes you need the computer to slow down a moment ("take a chill pill," as my students like to say) so you have the chance to interact. There is, indeed, a way, for even the shortest of programs to allow you to catch up with the computer.


The command SLEEP MAIN orders the computer to take a siesta break before carrying out whatever command(s) follow it in the main Task list.

If the only thing under the SLEEP MAIN command is the command EXIT PROGRAM, then you'll find you have as much time as you like to try out your click task. In fact, the computer will essentially be in an enchanted slumber that lasts either for eternity or until you abort your program by clicking on the Stop button on the Toolbar. This is not the most elegant of programming techniques (imagine a puppet show that would only end when the audience tore down the theater!), but it works just fine for really short and simple programs with one or two clickable objects.

As you begin to get into more complex programming, however, you'll find it more to your advantage to use SLEEP MAIN in conjunction with its opposite, WAKE MAIN. This command is usually found at the end of a sub-task, and causes the main task to "wake up" and carry out whatever instructions may follow the SLEEP

MAIN command leading up to the instruction to exit. (Often these post-snooze commands will take care of destroying any objects that you have in your program in order to free up the memory space that they consumed while your program was running.)

Keep WAKE MAIN in mind as you read on and learn more about branching out into sub-tasks, but for now, you can use the SLEEP MAIN trick by itself to give your clicker finger a break.

# UNIT 1
## Highlights

### COMMANDS:

MOVE OBJECT
MORPH OBJECT
CONSTRUCT OBJECT
PLACE OBJECT
SHOW OBJECT
DESTROY OBJECT
STORE BOX
LOAD BOX

SET LOOP
JUMP LOOP
JUMP ALWAYS
BRING-FRONT OBJECT
SEND-BACK OBJECT
HIDE OBJECT
RESIZE PLAYGROUND
INSTALL BACKGROUND
LOAD SOUND
PLAY SOUND
PLAY-N-WAIT SOUND
SLEEP MAIN
WAKE MAIN

### OTHER TOOLS:

Graphics Library
Sound Room
Object Designer
Program Window

### INSIDE THE COMPUTER:

Accumulator (AC)
Loop Register

# UNIT 2
## BRANCHING OUT:
## Creating Interactive Programs and Simple Games

Making animation is fun, but using your animation to get the user involved in interactive programs is even better! In this unit you'll learn a variety of skills and commands that will allow you to bring your users into the action. Of course this will make things a little more complicated as you learn to account for all the possible responses your users may enter. So we'll also be learning how to control for different possibilities, and break complex programs down into more manageable chunks.

In chapter 9 you'll learn commands for displaying and receiving text in the Conversation window, so you can write programs which "chat" with your users.

In chapter 10 you'll get an introduction to working with number data in FM, and learn to see boxes as variables that allow for flexibility and good style in your programs.

In chapter 11 you'll learn commands for checking to see if two moving objects are touching, so you can build different outcomes into your programs depending upon what's going on in the Playground window.

In chapter 12 you'll learn how to use commands that compare a whole bunch of different things in order to allow you to check for different possible conditions as your program runs its course, and build in corresponding outcomes.

In chapter 13 you'll expand your understanding of sub-tasks to see that they have many more uses in programming than just to make click-tasks for objects.

# CHAPTER 9

## Adding Text Strings to Your Programs for More Interactive Fun



Commands Introduced:

## ----- Teacher's Journal -----

Manuel and Andre were pooling their resources. They had begun in a feud because each boy claimed the other was copying his idea to make a speed boat program. But within minutes of working independently, Andre noticed Manuel's interesting, top-view art work, while Manuel became intrigued with Andre's use of a click-task to create an explosion. Now, they were huddled together in front of the same computer, a collaborative effort well underway.

They were almost finished when Andre thought of a problem.

"Wait," he said. "How they gonna know to click on the boat?"

"We tell 'em, dummy!" Manuel said, test-running the explosion one more time.

"I KNOW," Andre protested. "But I mean, you know, like if we're not here or something."

"What do you mean, if we're not here…" Manuel began.

"What he means…," I interjected (reconfirming my theory that all good teaching arises from good eavesdropping), "What he MEANS is, what if you guys decide to sell this game for a million dollars to some software company and they distribute it all over the place and someone in New York buys it in a store and takes it home to play with it…HOW ARE THEY GOING TO KNOW TO CLICK ON THE BOAT?"

"Yeah!" said Andre.

"Oh." said Manuel.

"Let me tell you about strings." I said.

## Communicating with Your Users

Once I had taught them to use strings, Andre and Manuel were able to stand back, proudly mute, as their program users read the ominous message WHATEVER YOU DO, DON'T CLICK ON THE BOAT! The text was automatically displayed in the Conversation window, just underneath the animation in the Playground.

After another lesson on strings, Jaime got busy transferring his bad attitude and rude wit into a humorous, grouchy computer persona:

"Hey, get over here! What's your name?" the Conversation window on his computer beckoned passers-by.

"Janet," I typed in response.

"Well, JANET," the computer replied, "That's a pretty sorry name! How old are you?"

"33," I responded.

"33?????? Well, are you ugly? Cause if you are, you better back off or I will self-destruct!"

The next participants got the same bad treatment, customized with their own names and ages worked into the computer's seemingly spontaneous responses.

You get an A+ if you've already figured out how to use the Sound Room and SOUND commands to enhance your programs with messages and instructions. But working with a lot of recorded sounds can get unwieldy, especially if you want to have a fairly involved "conversation" with the user. For this, and other reasons, you will often prefer to use text to communicate with your program users. For that, you need string commands.

I know what you're thinking. Some of these techie terms can get pretty weird. (Just wait until we get to "gaggles" and "booleans" in the next unit!) This string business really has nothing to do with kites or spaghetti.

The term "string" refers to any series of keyboard characters that you type in as input. It could be poetry, shopping lists, or just plain nonsense; to the computer, it's all just "strings" of textual characters.

✔LOAD STRING
✔WRITE-SCREEN STRING
✔The Backslash [\]

LOAD STRING and LOAD-SCREEN STRING and WRITE-SCREEN STRING are the two commands you need to communicate with your program users in the Conversation window.

When you use the command LOAD STRING, the Data Wizard will present you with a data-entry field where you can type in the string you want to use. You can edit your text in the text-entry field using standard word processing techniques for highlighting, inserting, and deleting text.

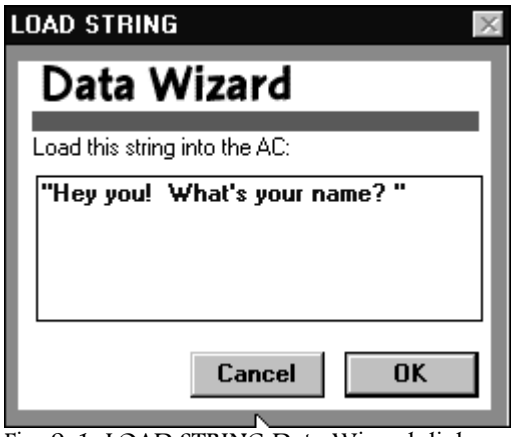


Fig. 9-1 LOAD STRING Data Wizard dialog

You can type in a message of up to 1,024 characters. The main thing to remember, if you want to keep from going crazy with strings, is this: String text must be between quotes, or the computer will not recognize it as a string! (I can almost guarantee that this will trip you up at least once before you remember it forever!)

Like CONSTRUCT OBJECT, the LOAD STRING command serves only to put the specified string into the AC where it can be further manipulated according to your needs.

If you want the user to be able to see the string printed up in the Conversation window, (which is always convenient for communication) you must use the command WRITE-SCREEN STRING. These two commands are all you need to post a message in the Conversation window. Let's play around with them before we look at how to let the user "talk back."

Unless you tell it to, the computer will print out all the input strings in one long line. If you want the computer to insert a line break, you use the backslash key on your keyboard to insert a thing that looks like this: \ . Every time it comes to one of those, the computer will

start a new line in the Conversation window. Just like any other string data, this character must be between quotes. If you want to make a carriage return at the end of a line of text, then just include the backslash at the end of the string like this: "Backslashes are for bozos!\". A backslash can also stand alone as its own string to make a blank line in the text you want to print out in the Conversation window. It looks like this in the code: LOAD STRING "\". You'll get a better feel for how this works once you've had a chance to play around with it a little bit.

## LET'S PROGRAM!

1. Go back to Myprog1.fmp. Choose Open Existing Programs, and navigate through c:\Fundam\Manual\Practice until you find the folder for this program.

2. Use LOAD STRING and WRITE-SCREEN STRING to add instructions, captions and quips. Insert two lines near the beginning of the program, and use the commands LOAD STRING and WRITE-SCREEN STRING to add some printed instructions for your user. Don't forget to include backslashes at the end of each instruction, so they'll each be on a separate line when they're printed out in the Conversation window.

3. Add strings to your sub-tasks. Spice up what happens when the user follows your directions and clicks the mouse. You'll need to get into the code for the sub-task "morph" for this. You can get there by pulling down the Window menu on the Toolbar, and selecting theProgram window from the list. In the Program window, double-click on the task name "morph," which appears in the list under main. That will automatically open up the Task window for this sub-task. You can add to and revise the code here in the same way that you do in the main Task window. When you're finished, you can get back to the main Task window via the Program window once again.

## Programming Tips and Tricks

You can type almost anything you want in the data-entry field on the LOAD STRING Data Wizard, but whatever you do... "DON'T FORGET THE QUOTATION MARKS!!!"

And remember, just because you told the computer to "think" of a particular message with your LOAD command, the computer won't remember to write the message on the screen unless you tell it to do so.

## Be Your Own Stage Hand

Once you've got stuff going on in both the Conversation and the Playground window, you may want to have some control over how they're placed, in relationship to each other, on the monitor screen. In the last unit, we learned about the command RESIZE PLAYGROUND. Here are a few more commands that work along similar lines.

✔PLACE PLAYGROUND
✔PLACE CONVERSATION
✔RESIZE CONVERSATION

Let's start with the last one first. RESIZE CONVERSATION works exactly the same way as RESIZE PLAYGROUND. It allows you to decide how big or small the Conversation window will be while your program is running. When you use this command, you'll see a familiar Data Wizard dialog, inviting you to specify dimensions for the window's width and height.

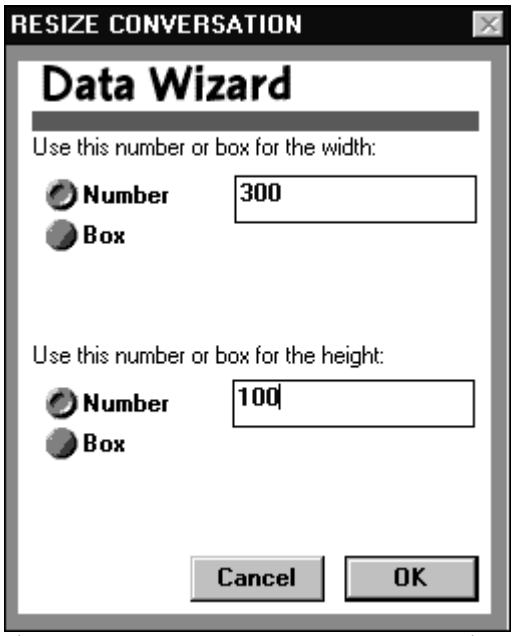



Fig. 9-2 RESIZE CONVERSATION Data Wizard dialog

The PLACE PLAYGROUND and PLACE CONVERSATION commands assume that a grid, similar to that used within the boundaries of the Playground window, is actually superimposed across your entire screen. You can assume that the point (0,0) is in the upper left-hand corner of your screen, and then place the two windows where ever you want them to be. Note that what you are really placing here are the windows' upper left-hand corners. You'll work with the Data Wizard here again, and plug in the coordinates as you wish. (Most often, you'll choose a zero for your x coordinate since it's nice to have the left edges of the Playground and Conversation windows

align with the left edge of the screen. But, as always, it's entirely up to you!)

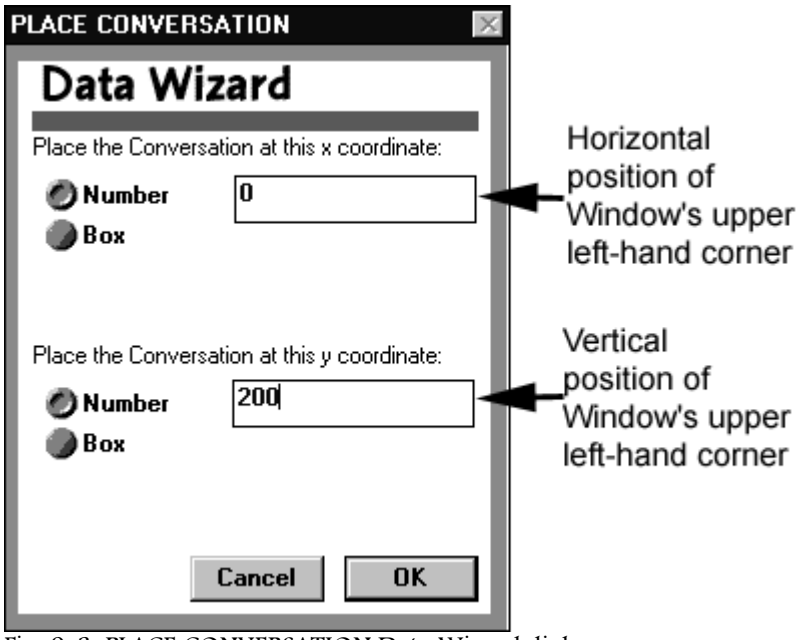



Fig. 9-3 PLACE CONVERSATION Data Wizard dialog

## Letting the User Talk Back

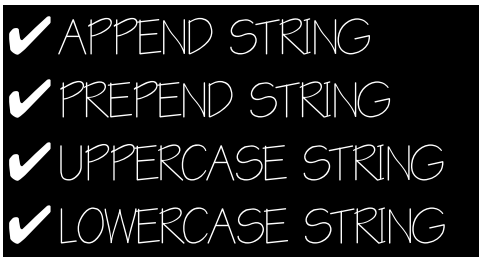
LOAD STRING and WRITE-SCREEN STRING are all you really need to write out simple messages and instructions to users of your animation programs. But what if the string that you load is not a statement, but a question? What if, like Jaime, you want the user to be able to answer back?

### ✔READ-SCREEN STRING

With just one extra command, strings can become the main ingredient for a whole different kind of interactive program. READ-SCREEN STRING causes the computer to pause in the course of the program and "read in" (to the AC) a message that the program user types on the keyboard. That's how Jaime got his computer to insult *my* name the first time, and Yolanda's the next.

## Spicing Up the Conversation

Once you or your user have put a string into the AC, there are several commands you can use that add to, or change, the text before you print it out to the Conversation window.




Instead of putting a new string into the AC (as do LOAD STRING and READ-SCREEN STRING), APPEND STRING adds something to the end of whatever string is in the AC at the time the APPEND command is executed. Let's say for example you have a program that asks the user to type in his name, and you want the computer to answer back with "_____ is the funniest name I ever heard of!" You can use the APPEND STRING command after READ-SCREEN STRING to tack that message onto whatever name the user types into the AC.

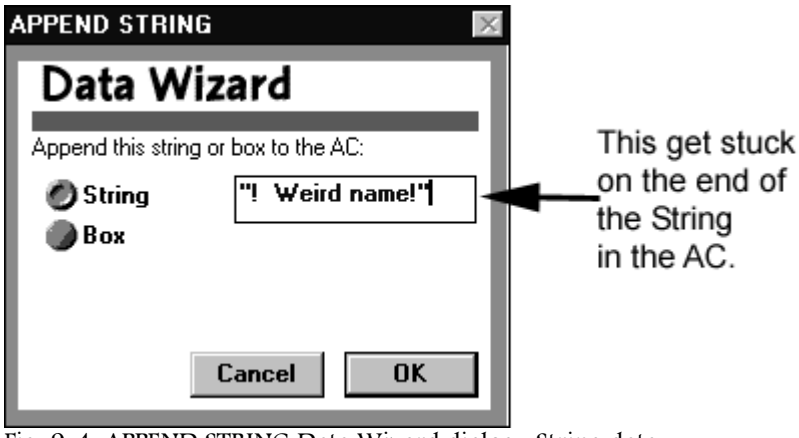



Fig. 9-4 APPEND STRING Data Wizard dialog. String data entered here gets appended to the string in the AC.

(Notice this dialog offers you a box option. We'll be discussing the use of boxes with strings later in the chapter.)

PREPEND STRING works the same way, only it tacks something onto the front of the string in the AC, as its name suggests. ("I never heard of a name like_____!")

UPPERCASE STRING and LOWERCASE STRING are two commands that you can use to further control the tone and appearance of the text on the Conversation window. The first makes all the characters in the string uppercase, and the second makes them all lowercase. Remember that these commands will only affect the string that is currently in the AC.

## LET'S PROGRAM!

1. Create a new program and name it heyyou.fmp. Start at the FM Welcome window, and click on the Create a New Program button. (Check back in chapter Six if you need a reminder of how to do this.) You don't have to worry about stocking your Graphics Library for this one until later...and that's only if you want to; strings can stand all on their own to make fun and funny programs.

2. Write a whole interactive conversation with string commands. Use READ-SCREEN STRING with APPEND and PREPEND STRING to give the impression that the computer is actually reading and responding to user input. Here's what Jaime's program looked like, to give you an idea of how to begin:

   LOAD STRING "Hey, get over here! What's your name?"
   WRITE-SCREEN STRING
   READ-SCREEN STRING
   UPPERCASE STRING
   PREPEND STRING "Well,"
   WRITE-SCREEN STRING
   LOAD STRING "That's a pretty sorry name! How old are you?"
   WRITE-SCREEN STRING
   READ-SCREEN STRING
   APPEND STRING "??????? Well, are you ugly because if you are…"

### Programming Tips and Tricks

Try thinking of your computer as a puppet for this assignment. Give it a character and a "voice." If you're creative, this simple program model is good for hours of fun!

## Using Strings with Boxes...

APPEND and PREPEND STRING let you tack things onto the beginning and the end of the string that's currently in the AC. But what if you want to hold onto a string and use it again later? Let's say you want to save up a bunch of user responses and print them all out at the end of the conversation. What commands do you think you can use to store away your strings in a place where you can get to them again?

You guessed it! You can use the same STORE BOX/LOAD BOX commands with strings that we used to store and reuse objects in the last unit. (See chapter 4 for reminders about using boxes to hold onto things for later use.) Handy little things, those boxes. Don't forget to name your string boxes appropriately so you can remember what's in them.

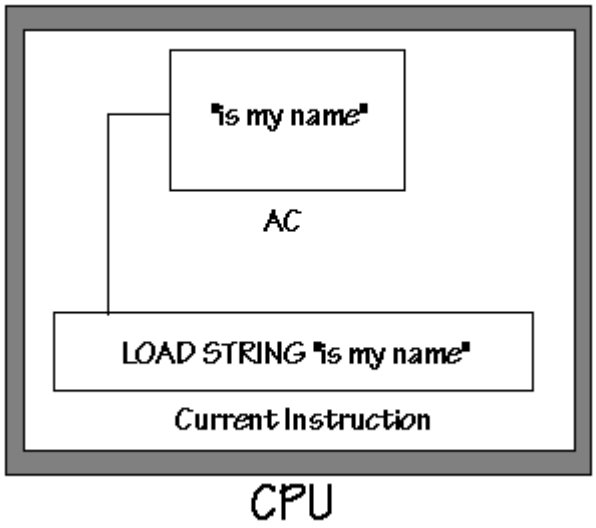
Remember MadLibs? Look at the following code sample. Can you tell what happens when this program is run?

```
LOAD STRING "What's your name?\"
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE BOX username

PREPEND STRING: "Cool name,"
APPEND STRING ".\"
WRITE-SCREEN STRING

LOAD STRING "What's your favorite color?\"
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE BOX usercolor

LOAD STRING "what's your favorite animal,"
APPEND STRING username
APPEND STRING "?\"
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE BOX useranimal

LOAD STRING "Hey, everyone!\"
UPPERCASE STRING
WRITE-SCREEN STRING
LOAD BOX username
APPEND STRING " has a "
WRITE-SCREEN STRING
LOAD BOX usercolor
APPEND STRING ""
WRITE-SCREEN STRING
LOAD BOX user animal
WRITE-SCREEN STRING
LOAD STRING "  in the back yard!\"
```

```
WRITE-SCREEN STRING

LOAD STRING "Thanks for playing, "
APPEND STRING username
WRITE-SCREEN STRING
```

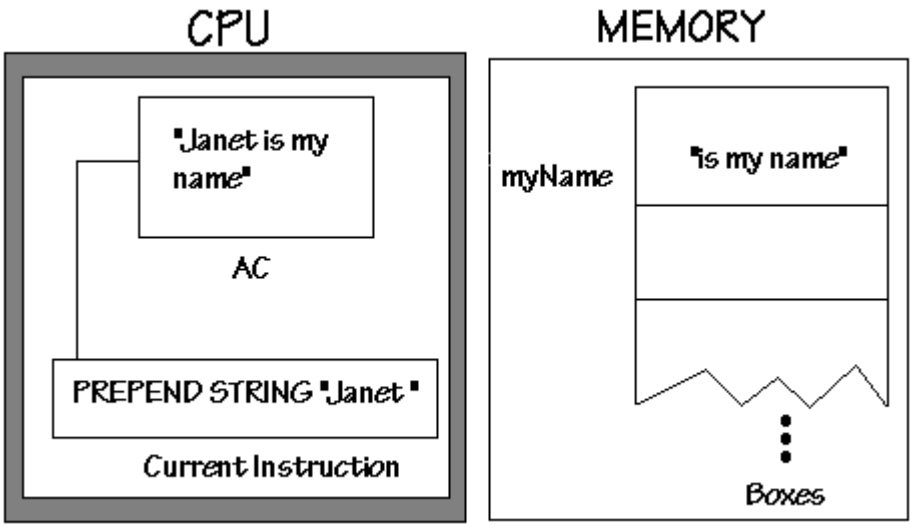
## What's Going On Inside the Computer?

This is a good time to revisit our old friend the AC (not that it hasn't been with us all along!). Strings are a great way to demonstrate how the AC operates. LOAD STRING and READ-SCREEN STRING are the two basic commands for putting a string into the AC.




Illus. 9A LOAD STRING

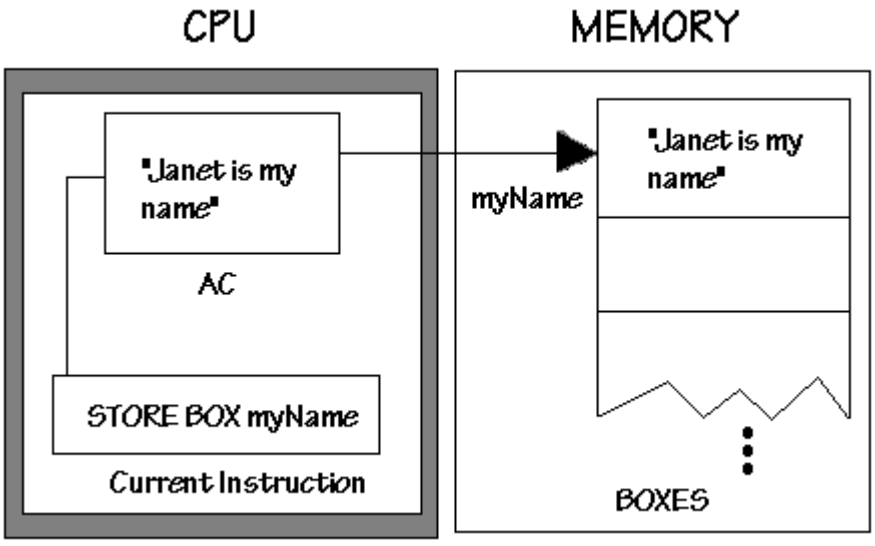
Whenever a string is loaded into the AC by you, or "read in" by the computer from something the user types in, then that and only that string is what gets changed with any APPENDS or PREPENDS, or whatever follows.




Illus. 9B PREPEND STRING

Once you use a LOAD STRING or READ-SCREEN STRING command again, whatever string was once there is now gone. Any commands that follow will apply to the new string, at least until it, too, is replaced by another.

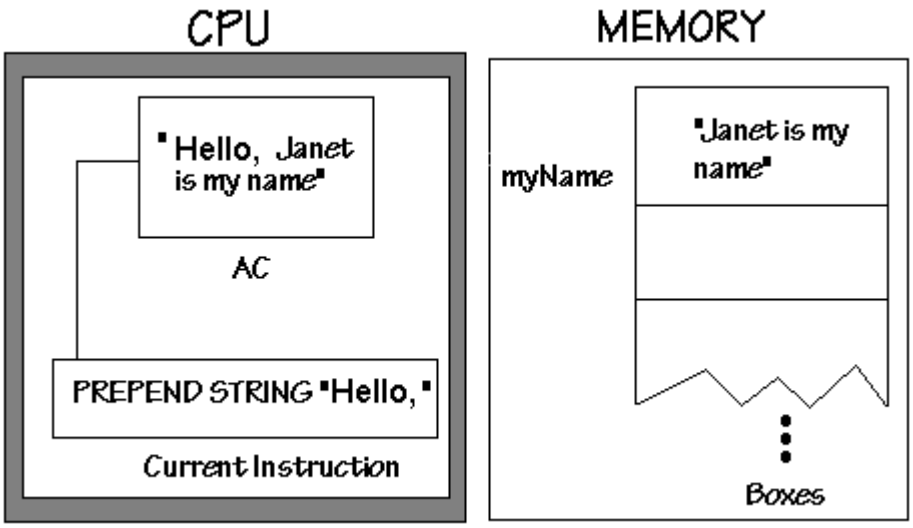
Like an object, however, a string can be copied into storage and then recopied back into the AC later in your program.




Illus. 9C STORE BOX copies the string currently in the AC into memory

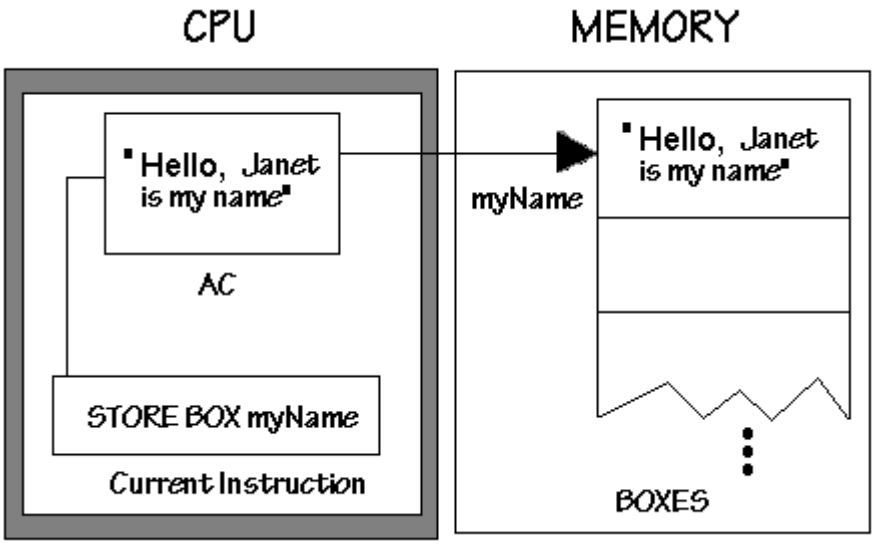
But strings have slightly different store/load rules than objects do. If you copy a stored object back into the AC, and then make changes to the object, the stored copy is automatically updated as well.

However, if you copy a stored string back into the AC, and then make changes to that string, the copy in storage remains unchanged.




Illus. 9D Stored string keeps its original form after AC copy has been changed

If for some reason you want to keep a copy of the changed string, you have to use the STORE BOX command again, so that the box will now contain the updated version.




Illus. 9E Repeating the STORE BOX command after changing a string puts the most current version in the box.

## LET'S PROGRAM!

1. Get back into the program called heyyou.fmp which you created earlier in the chapter. If you're starting a new session with this assignment, then you'll want to click on the Open Existing Programs button in the Welcome window: (c:\fundam\heyyou).

2. Use the sample program from this chapter as a guide to create your own MadLibs program using boxes and strings.

## Programming Tips and Tricks

"Janet has ten purple cows in the backyard!"

To print out the punch-line of your MadLibs program, you'll need to use a bunch of LOAD BOX commands instead of specifying actual text. To do this, you'll need to click on the little round "radio button" next to the word "box" that appears in the LOAD BOX Data Wizard. When the list of boxes comes into view, double-click on the name of the box that contains the string you want to load. And note that there is a box option for the APPEND and PREPEND commands as well. You can put a message like "My favorite color is also," into the AC, and then APPEND STRING with the box "usercolor."

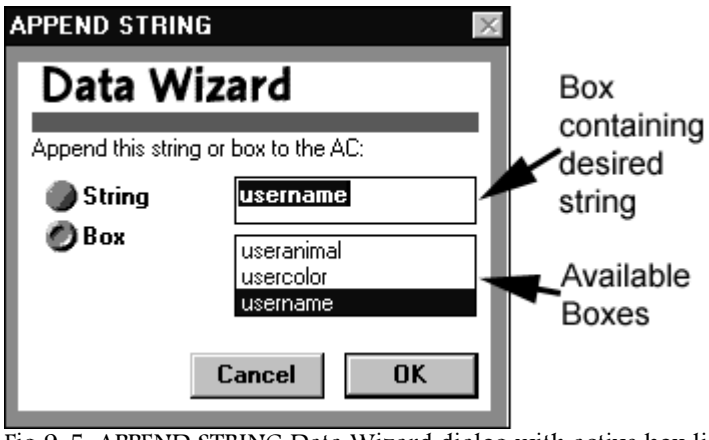



Fig.9-5 APPEND STRING Data Wizard dialog with active box list

## A New Take on Boxes

Notice that we're using boxes in a slightly different way here than we did when we used them with objects in chapter 4. Unlike the boxes we used to store Red Ridinghood and Wolf objects, these string boxes for MadLibs are used as place holders for data to which we don't have access when we are writing the program. When the computer

asks for and then reads in a user's name, for example, the name that then gets stored in the box "username" could be Annie, Albert, or Aloysius. In any case, the box is there, ready and waiting to hold the desired name. When used in this way, boxes serve as programming *variables*. Read on to chapter 10 for more discussion on how variables can add flexibility and elegance to your programs.

# CHAPTER 10
## Number Crunching and Variables



Commands Introduced:

| | |
|---|---|
| ✔ LOAD NUMBER | 133 |
| ✔ WRITE-SCREEN NUMBER | 133 |
| ✔ READ-SCREEN NUMBER | 133 |
| ✔ ADD NUMBER | 133 |
| ✔ SUBTRACT NUMBER | 133 |
| ✔ MULTIPLY NUMBER | 133 |
| ✔ DIVIDE NUMBER | 133 |
| ✔ REMAINDER NUMBER | 133 |
| ✔ RANDOM NUMBER | 136 |
| ✔ COMPARE NUMBER | 136 |
| ✔ JUMP= | 136 |

## ----- Teacher's Journal -----

Jose was giving me a preview of his program before doing a demo for his classmates. He clicked the Play button and I was treated to a short animation depicting a race between three little frogs. At the start of the program, his own voice came out of the computer's speakers: "On your mark…Get set…Go!"

"Cool!" I said, when frog number one crossed the finish line. He grinned up at me for a moment, but then his look of pride faded.

"It's boring!" he said.

"You really think so?" I asked.

"Yeah. The same guy wins every time. Look." He clicked the Play button once again to illustrate his point. As he'd predicted, frog number one was victorious once more. "See?" he said.

"It's still fun, though," I insisted.

"I guess so. But I wish…I wish there was a way for it to be a surprise each time. I wish it could be like the horse races, you know? Like…"

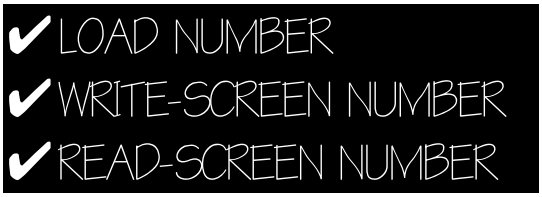
"Random?" I filled in, feeling a new lesson coming on.

"Yeah!" he said. "That's it– *random*."

"Well guess what. There is a way. There's a command called RANDOM NUMBER that I think could help you out here. Let me show you how it works…"

# "I'm Thinking of a Number..."

Now that you've had a chance to play around with the ABC's of FUNdaMENTAL, it's time to think about what you can do with 1,2,3 (and any other integer). numbers are a big deal in programming. You can find them in plain view in programs that ask the user for some kind of numerical input to reinforce math skills. But number commands are also hidden in a lot of the dynamics for more complex animation and gaming programs. Let's take a look at some number commands.




We've seen that the AC can contain objects, sounds, and strings. It can also contain numbers. LOAD NUMBER and READ-SCREEN NUMBER are the two most direct ways that numbers can get into the AC. With the first, you specify which number is loaded into the AC.

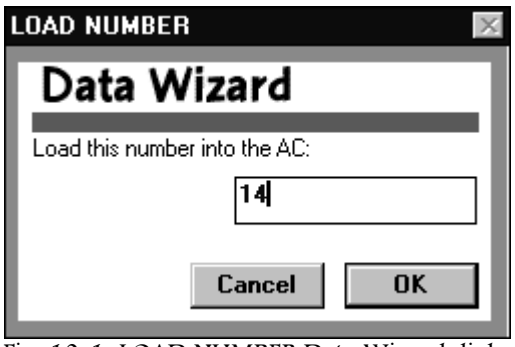
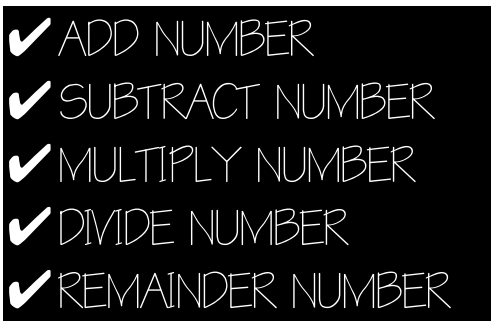



Fig. 10-1 LOAD NUMBER Data Wizard dialog

With the second, you can allow your program user to specify the number in the AC. WRITE-SCREEN NUMBER allows the user to see whatever number the computer is currently "thinking of." Once you have a number in the AC, there are several commands you can use to manipulate it.




You can use a variety of commands to do basic arithmetic with the number in the AC.

When you use ADD NUMBER, the Data Wizard will ask you to specify the number that you would like to have added to the number in the AC.

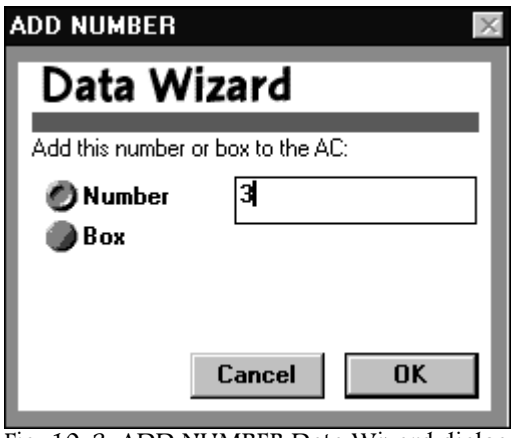


Fig. 10-2 ADD NUMBER Data Wizard dialog

The result will be a line in your program code that looks something like this:

ADD NUMBER 3

If the AC contained a 7 before coming to the instruction above, it would contain a 10 after completing it.

SUBTRACT NUMBER, MULTIPLY NUMBER, and DIVIDE NUMBER all work the same way.

REMAINDER NUMBER divides the number in the AC by the input number, and then puts the remainder in the AC. If the AC contains a 10 and you use the instruction REMAINDER NUMBER 7, the AC will now contain a 3l.

Stop and think for a moment about how you might be able to use these commands to make fun and interesting programs. Can you imagine, for example, how this program would run?:

```
LOAD STRING “Hi, what’s your name?/”
WRITE-SCREEN STRING
READ-SCREEN STRING
PREPEND STRING “How old are you, “
APPEND STRING “?\”
WRITE-SCREEN STRING
READ-SCREEN NUMBER
STORE BOX userage

LOAD STRING “Wow! Gettin’ up there!\ And by the year 2050,
you’ll really be over the hill!\You’ll be,”
WRITE-SCREEN STRING
LOAD BOX userage
ADD NUMBER 53
```

```
WRITE-SCREEN NUMBER
LOAD STRING "!!!!!!!"
WRITE-SCREEN STRING
```

## What's Going On Inside the Computer?

And as you're thinking…keep in mind that numbers, too, can be stored in boxes using the same STORE BOX and LOAD BOX commands that you used for strings and objects. As always, you'll want to give your box a good name, so you can later distinguish it from the other boxes in storage. We'll use the name "numbox" (for "number box") as our example here, but you'll probably want to be even more specific in naming the boxes for numbers in your programs.

The rules for stored numbers are the same as the rules for stored strings: If you copy a stored number from "numbox" back into the AC, and then change that number somehow, the copy in "numbox" remains unchanged. If you want to keep a copy of the new number, you have to use the STORE BOX command again to put the updated version of the number into "numbox." The code you need to use looks like the example below:

```
LOAD BOX numbox
ADD NUMBER 3
STORE BOX numbox
```

That way, the sum of numbox plus 3 is saved as the new contents of numbox.

## LET'S PROGRAM!

1. Go back to (or create) a program which uses strings to interact with the user.

2. Add a string that elicits some kind of numerical input from the user. Something like this will do nicely:

   ```
   LOAD STRING "How many buttons are on your clothes?"
   WRITE-SCREEN STRING
   READ-SCREEN NUMBER
   ```

   (Anything goes here: age, height, number of siblings, etc.)

3. Use STORE BOX to make sure you can get back to the number if you need it later.

4. Now use the MULTIPLY NUMBER command, and tell the user how many buttons there would be if she were standing next to 42 identically dressed friends. (Of course the 42 is just an example. You can multiply by any number; maybe you'd like to try something a little bigger, like 420? 4,200?

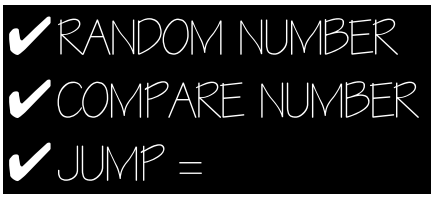
5. Run the program using your own stats as the "usernumber" data, just to get a feel for how the commands get processed.

## Programming Tips and Tricks

Play around with the number using all of the NUMBER commands in FM. Take your time and have fun. Tell the user how many buttons she and 42 identically dressed friends would be wearing altogether, or how old she'll be in the year 2050, or how much less he weighs than the average adult elephant, or how far 10 of her clones would reach if they lay down end to end...

## More Number Fun

There are two more things that FUNdaMENTAL can do with numbers in the AC which are both useful for adding elements of surprise and complexity to your programs. As Jose found out, you can get a lot of use out of numbers that are chosen randomly by the computer. And you can also use numbers as the basis for making comparisons and determining different outcomes for different possibilities.




RANDOM NUMBER takes two numbers or boxes as input, and then randomly chooses a number between them, putting the resulting number in the AC. When you use the RANDOM NUMBER command, the Data Wizard will ask you to input two numbers, two boxes, or one of each, to set the range the computer will then choose from (kind of like making a customized die or roulette wheel!)

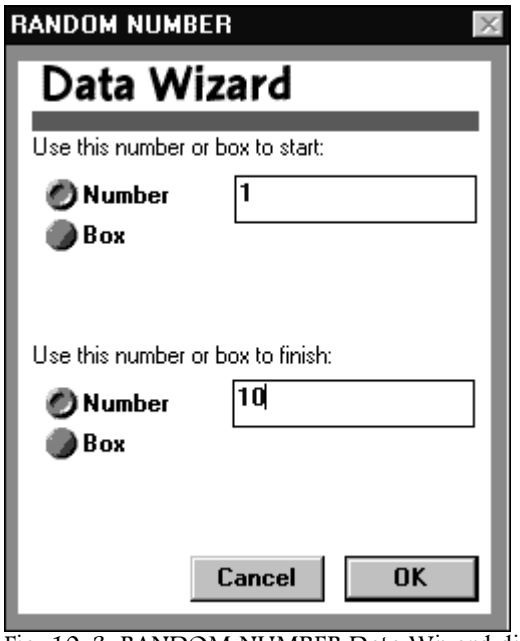


Fig. 10-3 RANDOM NUMBER Data Wizard dialog

COMPARE NUMBER tells the computer to compare whatever number is currently in the AC with the number or box that you input. The computer will be able to tell when two numbers are equal (for more information about how it accomplishes this, see chapter 11), and you can use that to your advantage in many different ways in your programs.

The COMPARE NUMBER command is often paired with JUMP =. This command, in the same jumping family as JUMP LOOP and JUMP ALWAYS, causes the computer to jump to a different place in the code if two things that are being compared are, indeed, equal. Just like its JUMPing counterparts, the command JUMP = has to have a marker in order to mark the spot in the code to which the computer needs to jump.

Look at the following sample, and take special note of how boxes are used:

```
LOAD STRING "Hi, how old are you? (Don't worry, I won't tell!) \"
WRITE-SCREEN STRING
READ-SCREEN NUMBER
STORE BOX userage

LOAD STRING "About how many inches tall are you?\"
WRITE-SCREEN STRING
READ-SCREEN NUMBER
STORE BOX userheight

RANDOM NUMBER userage, userheight
STORE BOX secretnumber

LOAD STRING "I'm thinking of a number between…"
WRITE-SCREEN STRING
```

```
    LOAD BOX userage
    WRITE-SCREEN NUMBER

    LOAD SCREEN STRING "and"
    WRITE-SCREEN STRING
    LOAD BOX userheight
    WRITE-SCREEN NUMBER
    LOAD STRING "Can you guess what it is?  You have three
                chances...\"
    WRITE-SCREEN STRING

    SET LOOP 3
 @guess
    READ-SCREEN NUMBER
    COMPARE NUMBER secretnumber
    JUMP = @congrats
    LOAD STRING "Nope!  Try again!\"
    WRITE SCREEN STRING
    JUMP LOOP @guess

    JUMP ALWAYS @bye
@congrats
    LOAD STRING "Congratulations!  You got it!!!!!  Pick up your
                prize at the door..."
    WRITE-SCREEN STRING
 @bye
    LOAD STRING "Thanks for playing!"
    WRITE SCREEN STRING

    EXIT PROGRAM
```

Can you tell what this little program does when you run it? If not, go ahead and type it in yourself. Then run it and see!

## Boxes as Variables

Did you notice how boxes came into play in the above code sequence? Just like their data counterparts (objects, strings, and sounds), numbers can also be stored away in boxes, and the contents of a box can then be used as the input for a subsequent NUMBER command. For example, if you have a program that asks the user to type in her age, and you store that number in a box called "userAge," you can later put another number into the AC and then use the command MULTIPLY NUMBER userage in order to multiply the AC number by the age of your program's current user.

To choose a box for such an operation, instead of specifying a number, you first need to click on the little round "radio button" next to the word "box" in the Data Wizard dialog.

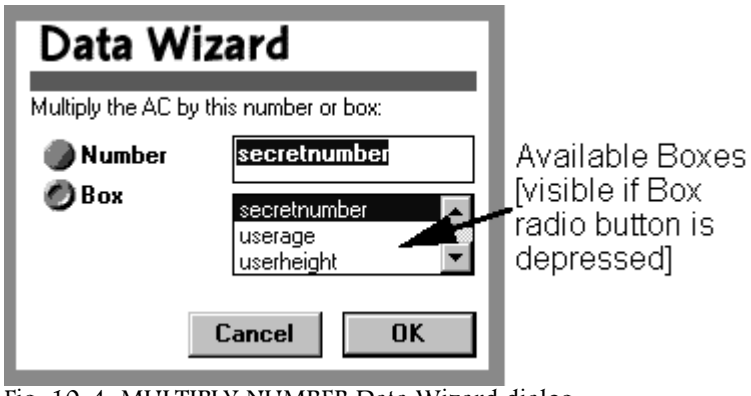



Fig. 10-4 MULTIPLY NUMBER Data Wizard dialog

That will cause a list of the boxes currently in storage to appear. You can select an existing box by clicking on its name, or you can type a box name straight into the data-entry field, if you don't yet have any boxes in storage.

Using boxes in this manner allows you to account for variables in your programming. Variables are essentially placeholders for certain types of data. In the first sequence of commands in the previous sample code, the computer is told to elicit a number representing the age of the user, and to store the resulting number in a box called "userage." The type of data (a number representing age) will always be the same, but the specifics will vary with each user (hence the term "variable").

In the early chapters on objects, we used boxes as a way to save a particular (or constant) piece of data so we could get back to it again later. We knew in advance, for example, that the Red Ridinghood and Wolf objects would be stored in the boxes we defined for them.

But if you focus on the box and not so much on the particular data that goes in it, you'll really be thinking like a programmer. And numbers are a good place to start, since many of us are accustomed to thinking about variables as part of the realm of mathematics.

So instead of saying to yourself, "I want the frog to hop a distance of 150," you say, "I want the frog to hop *some distance*." Then, somewhere else in your program, use commands that put some number into the AC, and follow that with STORE BOX somedistance.

It's sort of like leaving a space in your schedule to talk to students' parents every Thursday from 3:00 to 3:30. The parents you'll be speaking with each Thursday afternoon will vary, but the time slot is set up in advance so you're always ready to make the necessary calls.

Boxes can hold spaces for every kind of data in FM. You were using boxes as variables in the last chapter when you stored away string input from the user to make a MadLibs game, using commands like STORE BOX usercolor. In the same way, you can use boxes to mark the place where you want *some* sound or *some* object to come into play.

# Looking at Some Old Commands in New Ways

Knowing that you can use boxes, or variables, instead of constant numbers, you can teach a couple of old commands some new tricks.

## Set Loop

Look at the code sample below. How is SET LOOP affected by the use of a box instead of a constant number?

```
    LOAD STRING "So, it's your birthday...how old are you?\"
    WRITE ~SCREEN STRING
    READ~SCREEN NUMBER
    STORE BOX userage

    LOAD STRING "Well, then…"
    WRITE~SCREEN STRING

    SET LOOP userage
@happy!
    LOAD STRING, "Happy birthday!
    UPPERCASE STRING
    WRITE~SCREEN STRING
    JUMP LOOP @happy!

    LOAD STRING "And one to grow on!: HAPPY BIRTHDAY!"
    WRITE~SCREEN STRING

    EXIT PROGRAM
```

## Move Object

So far we've been looking at how number commands can spice up your conversation programs. But the real power of number commands comes out when you use them in your animation programs.

Imagine, for example, a game of chance in which a user must predict the winner of a frog-jumping contest. Assume that the following code sequence is preceded by the necessary commands to CONSTRUCT, PLACE, SHOW, and STORE BOX three frog racers.

Notice, again, how the command JUMP = causes the computer to jump to a different place in the code if two things being compared are equal. Otherwise (if they're *unequal*), the computer just skips over the JUMP = and goes on to the next command in the list.

```
@random
   RANDOM NUMBER 1,3
   COMPARE NUMBER 1
   JUMP = @go1
   COMPARE NUMBER 2
   JUMP = @go2
   COMPARE NUMBER 3
   JUMP = @go3
 @go1
   LOAD BOX frog1
   JUMP ALWAYS @hop
 @go2
   LOAD BOX frog2
   JUMP ALWAYS @hop
 @go3
   LOAD BOX frog3
   JUMP ALWAYS @hop

 @hop
   MORPH OBJECT legstraight
   MOVE OBJECT 0, 50
   MORPH OBJECT legsbent
   JUMP ALWAYS @random
```

## Place Object

And what happens when we use a randomly chosen number as the coordinate input for the PLACE OBJECT command? The following sample code shows the commands necessary to construct and store a target object called sittingduck. The intent of the program is to get the object to move around the Playground unpredictably to set up a sort of a shooting gallery game.

```
   RESIZE PLAYGROUND (350,300)
 @shift
   RANDOM NUMBER (0,350)
   STORE BOX leftsideobject
   RANDOM NUMBER (0, 300)
   STORE BOX bottomobject

   LOAD BOX sittingduck
   PLACE OBJECT leftsideobject, bottomobject

   EXECUTE SUB-TASK wait
   JUMP ALWAYS @shift
```

EXECUTE SUB-TASK is what keeps the target from moving hopelessly fast. You'll learn how to use the commands to manipulate time lapses in your programs in the next unit. Go look up GET TICKS if you can't wait to wait!...)

# LET'S PROGRAM!

Becoming an advanced programmer means finding creative ways to build flexibility and efficiency into your thinking, and into your programs.

1. Open up one of your animation programs and get ready to use some familiar commands in new ways…

2. Use RANDOM NUMBER and STORE BOX to build in a customized game spinner somewhere near the start of the program. Do this by inserting two lines of code that look something like this:

   RANDOM NUMBER 1,10
   STORE BOX random#

3. Now use the "random#" box (instead of typed-in constants) as the input for a command like SET LOOP or MOVE OBJECT.

4. Use the sample code for the race program from earlier in this chapter as a model, and make your own race game, either by revising an old program or making a new one.

## Programming Tips and Tricks

Expert programmers plan out much of their code in advance, laying out their variables ahead of time by adding things directly to the Boxes region of the Task window. Here's how to do it:

1. Start at the top, or at the first empty space, of the box list, and click there. You'll see a highlighted area for entering text.

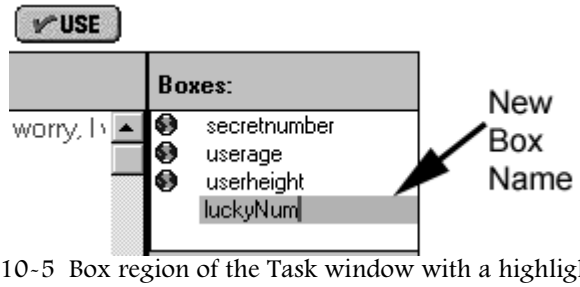



Fig. 10-5 Box region of the Task window with a highlighted line ready to receive a new box

1. Now, type the name of the box you would like to define and press the Enter key on your keyboard when you are done.

2. Add as many boxes as you think you'll need for the variables in your program.

Now, when you use the command STORE BOX, the computer will have a predefined variable to connect with the data you've selected. (You won't be seeing that little prompt asking, "Would you like to define this box?" because you'll have already defined it.

# CHAPTER 11
## Making a Splash with TOUCHING OBJECT



Commands Introduced:

## ----- Teacher's Journal -----

Remember LaToya? The girl who wouldn't leave the classroom? Well, she finally got her diver to march resolutely to the end of the diving platform (and no further!), bend over, and dive gracefully down into the cube of blue water underneath. I actually thought I might make my afternoon meeting, and was packing up my things, when…

"Wait!" she said, "What about a splash?"

"A splash?" I said, putting down my bag and taking off my watch.

"Yeah," she said, "when she hits the water, there should be a splash…"

## A Command, Indeed!

I missed my meeting so I could stay and teach LaToya about TOUCHING OBJECT. When I first told her the name of this command, she wrinkled up her nose and said, "Touching object? What kind of a command is that?"

I repeated it to myself and realized that it is, indeed, no kind of command at all…unless you understand it to be an abbreviation for "Check-to-see-if-[the object in the AC]-is touching-[this other]-object".

"You see," I explained, "it's really *checking,* and not touching, that you're telling the computer to do."

She rolled her eyes at the inconvenience of having to absorb all this, then quickly absorbed it and got on with the business of using the new command to make a splash.

### ✔TOUCHING OBJECT

When you use the command TOUCHING OBJECT, the Data Wizard will show you a list of all the program's stored objects so you can indicate which *other* object the computer should look for when checking to see if the AC object is touching something.

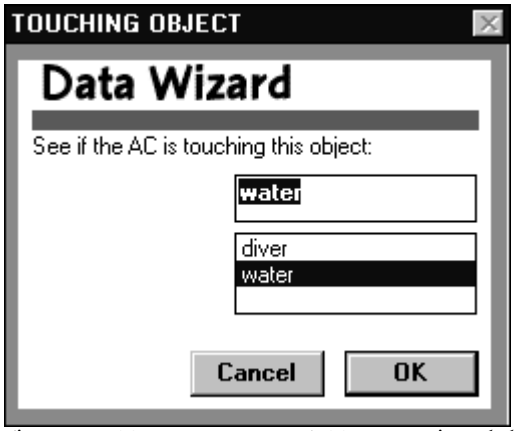



Fig. 11-1 TOUCHING OBJECT Data Wizard dialog

LaToya, for example wrote code that looked like this,

```
@dive
    MOVE OBJECT 1,-2
    TOUCHING OBJECT water
    JUMP ALWAYS @dive
```

Notice that she made the diver's descent slow and graceful by looping through a whole bunch of little (1,-2) moves. Then she used TOUCHING OBJECT in order to tell the computer this: "After every

downward move, check to see if the diver (which is in the AC) is touching this other object: water."

She placed this command directly after the MOVE command, which essentially causes the computer to ask– and answer– the question "Are we there yet?" over and over and over again, like a three-year-old in the back seat of the car.

Notice that the above sample code only accounts for the answer "NO!" If the answer is no, then we keep driving (or in this case diving), by jumping back up to move the diver down a bit more, before asking once again, "Are we there yet?" (Remember that the computer can whiz through your instructions so fast that all this moving and checking still translates into animation that appears quite fluid.)

But what about when the answer is finally "YES!"????

## "If we're there, then get out of the car; otherwise keep driving..."

Right now, the sample code above checks to see "if" the diver object is touching the water object, but it only accounts for the "otherwise" possibility by jumping always to the code that keeps her diving without morphing the water to make a splash.

We need to tell the computer to jump somewhere else if the diver is actually there! (Imagine the screams of protest from the back seat if, after an arduous trip, you finally did get to Grandma's, and just kept driving!!!!) It's time to use JUMP = again.

Look at this section of code and see if you can tell what's going on:

```
@dive
    MOVE OBJECT 1,-2
    TOUCHING OBJECT water
    JUMP = @splash
    JUMP ALWAYS @dive

@splash
    LOAD BOX waterbox
    MORPH OBJECT splshwtr

@finishdive
    LOAD BOX diverbox
    MOVE OBJECT 1, -2
    JUMP ALWAYS @finishdive
```

Notice how the JUMP = command works together with TOUCHING OBJECT. Remember that the TOUCHING OBJECT command is really a "checking" command, which means that the program must have

two possible outcomes built into it. As long as the diver is not yet touching the water, we want her to keep descending with the water left undisturbed. But once the diver does touch, we want something different to happen. We want the computer to jump down (in this case) and follow a whole new set of instructions that first creates a splash on the surface of the water and then makes sure the diver keeps descending through the water.

So JUMP = in this context roughly translates into, "If they are touching then go do what's after this marker..."

## What's Going On Inside The Computer?

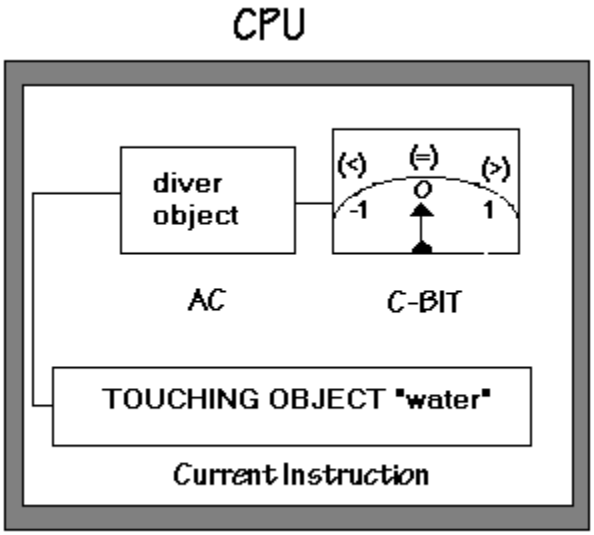
How does the computer really tell if two objects are touching? It can't look the way we do. Instead, it uses numbers. When LaToya used the command TOUCHING OBJECT, the computer didn't care about the diver and the water. All it did was consider the placement of both objects in the Playground window. As soon as any of the diver object's coordinates overlapped or equaled any of the water object's coordinates, the computer "knew" that the two objects were touching and followed the instructions for that case.

So each time the diver moved a tiny bit further down, the computer stopped and checked for overlapping coordinates. Whenever the computer checks anything, it needs to store the outcome of the comparison in the simplest, most direct terms possible. These terms will then be used to allow the JUMP command to do its thing (or not).

The AC is busy holding objects and their coordinates, so the outcome of each comparison has to be stored in another place called the Comparison bit (C-bit for short). You can think of the C-bit as something like a little dial gauge, with three possible settings: less than (<), greater than (>), and equals (=).

Each time the command TOUCHING OBJECT is executed, the C-bit gets reset in terms of these three values, according to the outcome of each comparison. As LaToya's diver dove, the C-bit settings went something like
"greaterthangreaterthangreaterthangreaterthanEQUALS!!!!!!!"
(Note that you can see the current C-bit reading at any time during your program's run time by looking into the Debugger window. The C-bit indicator will tell you the reading currently on the C-bit as you step through each execution of the TOUCHING OBJECT command.)

The command JUMP = doesn't know or care what is being compared. All it does is read the C-bit each time, waiting for that "equals" setting to give it the cue to jump.




Illus. 11A When two objects being checked by the TOUCHING OBJECT command touch, the C-bit is set to "equals" and JUMP = is executed.

## LET'S PROGRAM!

1. Open Program6.fmp. Go to the Welcome window and click on Open Existing Programs. Navigate through c:\, Fundam, Manual, Practice, and Program 6. Then open Program6.fmp.

2. Find the part of the code that looks like this:

   LOAD BOX frog
   @frog_jump
   MOVE OBJECT 4, 1
   JUMP ALWAYS @frog_jump

   As it is, the frog will just keep on flying past the princess, even after the frog_prince object touches the princess object.

3. Insert the TOUCHING OBJECT command so that, after each move of (4,1) the computer checks to see if the two objects are touching.

   In a moment, you'll need to put the command JUMP = directly after TOUCHING OBJECT (hint: to make room for these two commands in your code, insert two line spaces after MOVE OBJECT 4,1 and JUMP ALWAYS@frog_jump). But before you insert the JUMP =, you have to place a new marker somewhere else in the code.

4. Insert the marker @kiss. Insert a line after the JUMP ALWAYS command, and then click on the marker radio button above the Instruction list. Type in the label @kiss, and click OK.

   This marks the spot where you'll write the code for what happens when the princess and the frog prince kiss. We'll come back and write that section in just a minute.

5. Insert the JUMP = command just after TOUCHING OBJECT and select the @kiss label in the Data Wizard.

6. Write the code for what happens when the objects touch. Now comes the fun part: Return to the new marker, @kiss, and insert some lines after the marker. Write the code that will make the frog prince morph into a regular prince if the two objects are, indeed, touching.

## Programming Tips and Tricks

You'll definitely want to take a good look into your Graphics Library for this program in order to familiarize yourself with all the funny ".bmp" names to go with your different graphics. Make sure to check out "frprsmok.bmp" for a nice, transitional image to use in the frog's transformation. There's a fun sample solution for this program in your Examples folder.

Here's a challenge: Can you figure out a way to use a click-task so that the user will have some control over whether or not the two objects actually touch? Decide whether having the two objects touch is something the user wants to achieve or avoid, and... you have your first game! (If this feels a little overwhelming, we'll be revisiting click -tasks in chapter 13. Maybe you can try this then!)

# CHAPTER 12

## Bouncing Around: GET, COMPARE, and JUMP Commands



Commands Introduced:

## ----- Teacher's Journal -----

Tom had a good program going. A lot of kids were gathered around his computer watching as an alien and a puppy moved randomly around the Playground window. When the two objects happened to touch, the dog was transformed into an identical alien, and a spaceship appeared to take both aliens away.

Trouble was, it only worked about half the time. Since the numbers were chosen randomly, they sometimes caused the objects to move away from, rather than toward each other. In some cases, one or both of them simply disappeared off the edge of the screen.

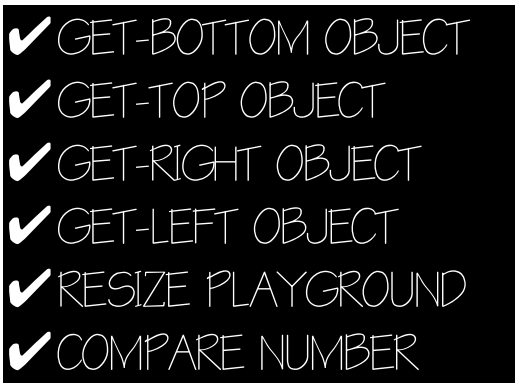
"There should be TOUCHING OBJECT for the top and sides of the Playground," Tom lamented. "Then they could, you know, bounce back the other way if they hit the edge. They'd have to touch each other sooner or later that way." He clicked on the Play button to try his luck at a successful run one more time.

"In a way, there is such a thing," I said. "It has to do with finding the coordinates of your objects as they move, and keeping track of where they are in relationship to the edge. It's a whole bunch of commands that all start with GET. Come on, I'll show you."

## Boink!

Let's say you want to write a program like the old Pong game, which allows users to hit a ball back and forth between two paddles. So far we've learned about how to use the TOUCHING OBJECT command as a way to program the computer to check if two objects–like, say, a paddle and a ball–are touching each other.

But, as Tom's problem illustrates, TOUCHING OBJECT wouldn't allow us to make sure that the computer "bounces" the ball off the side of the Playground to keep it in play, rather than just letting the ball fly out of view on the first missed shot. In this case, there's no other object to check, unless you've gone to the trouble of creating and importing some kind of boundary objects.




Let's forget about the paddles for a minute and just think about what it would take to keep a ball "bouncing off the walls of the Playground." The following code sample does just that. First, cover up the side notes and take a look at the code to see if you can figure out what's going on. Then you can check the notes to confirm your thoughts.

```
RESIZE PLAYGROUND 300,300          assures you know the
                                   playground dimensions, so
                                   you can use the numbers
                                   later with COMPARE NUMBER

CONSTRUCT OBJECT ball              constructs and stores the ball
PLACE OBJECT 150, 150
SHOW OBJECT
STORE BOX ballbox

RANDOM NUMBER -10, 10              picks a random num-
                                   ber between -10 and 10
```

```
    STORE BOX leftright                 stores the number to
                                        use as left/right move amount

    RANDOM NUMBER -10, 10               puts another random number
                                        in the AC

    STORE BOX updown                    stores the number to use as
                                        up/down move amount

@move                                   marks the code for when the
                                        ball is NOT touching walls

    LOAD BOX ball                       puts the ball in the AC

    MOVE OBJECT leftright, updown       moves it the amount determined
                                        by random number commands

    GET-LEFT OBJECT                     puts number corresponding to current
                                        left coordinate of object into the AC

    COMPARE NUMBER 0                    compares number with coordinate
                                        of Playground's left edge(0)

    JUMP <=@sidewallbounce              jumps to marker if left side of
                                        object is at 0, or less

    LOAD BOX ball                       puts the ball in the AC

    GET-RIGHT OBJECT                    puts object's current right
                                        coordinate in the AC

    COMPARE NUMBER 300                  compares right coordinate with
                                        coordinate of Playground's right edge
                                        (300)

    JUMP >=@sidewallbounce              jumps to marker if right side
                                        of object is a 300 or more

    LOAD BOX ball
    GET-TOP OBJECT
    COMPARE NUMBER 300
```

```
JUMP <=@topbotbounce

LOAD BOX ball
GET-BOTTOM OBJECT
COMPARE NUMBER 0
JUMP >= @topbotbounce

JUMP ALWAYS @move

@sidewallbounce
  LOAD BOX leftright                    loads distance of left right
                                        movement

  MULTIPLY NUMBER -1                    reverses direction of movement

  STORE BOX leftright                   stores movement with new
                                        direction to use in MOVE command

  JUMP ALWAYS @move                     returns to code that keeps
                                        object moving

@topbotbounce
  LOAD BOX updown                       reverses ball's direction for vertical
                                        moves
  MULTIPLY NUMBER-1
  STORE BOX updown
  JUMP ALWAYS @move
```

## Think Before You Jump...("If …, then …; otherwise, …")

Among the things you may have noticed in this elegant and efficient little piece of code are some interesting uses of JUMP commands. Let's take a moment to look at this.

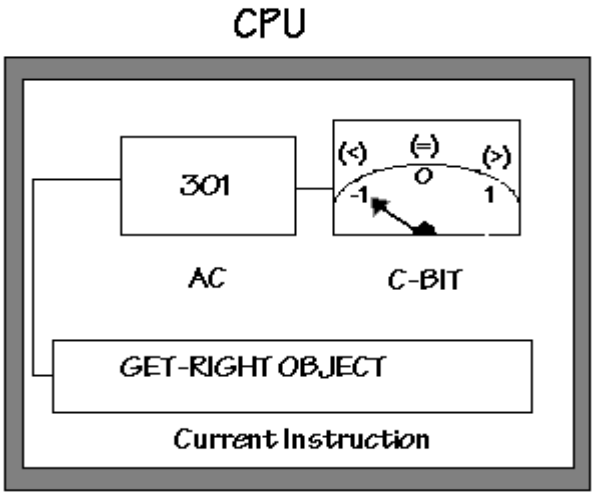
Whenever the computer makes a comparison, it's comparing a number associated with the contents of the AC, to another number. JUMP = accounts for the possibility that they are equal. There are a whole bunch more, one for every possible combination of C-bit readings:

- ✔ JUMP<= (jumps if the comparison results are 'AC is less than..' or ' AC equals')
- ✔ JUMP< (jumps if the comparison result is 'AC is less than')
- ✔ JUMP>= (jumps if the comparison result is ' AC is greater than' or 'AC equals')
- ✔ JUMP> (jumps if the comparison result is 'AC is greater than')
- ✔ JUMP< > (jumps if the comparison result is anything but 'AC equals')

## What's Going On Inside the Computer?

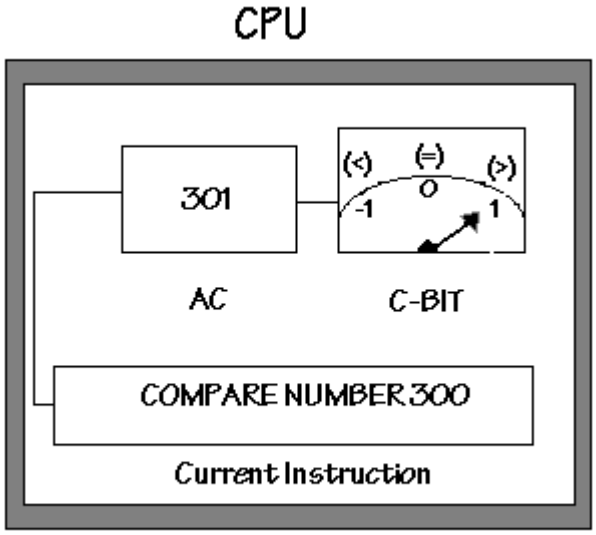
Remember the C-bit? That's the place that stores the results of any comparison that the computer makes. The C-bit has three settings: equals (=); less than (<); greater than (>). Depending upon what's going on, you may want the computer to jump to another part of the code if the C-bit shows something other than just equals. In the code sequence on the previous page, the programmer told the computer to jump if the C-bit read ">" OR "=." Why?

Remember, the computer doesn't "see" the picture of the ball as we do; it just keeps track of the whereabouts of a little cloud of dots called pixels. When we GET-LEFT OBJECT, we are telling the computer to load into the AC the number coordinate of the leftmost pixel of the ball object.



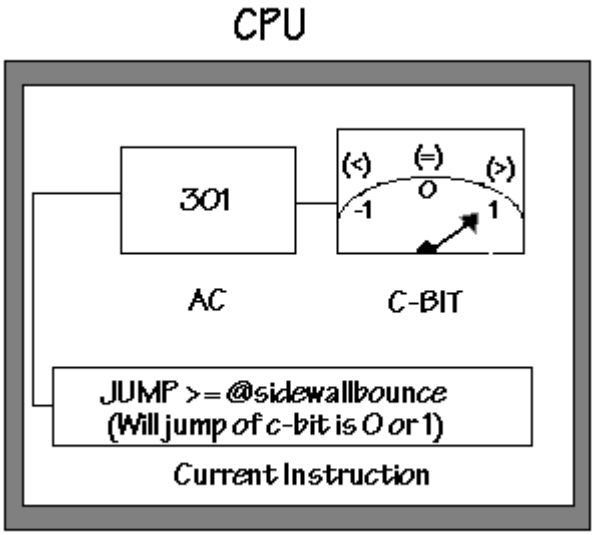

Illus. 12A GET-LEFT OBJECT

JUMP = would tell the computer to send the ball back the other way when the edge of the ball object exactly equaled the coordinate at the edge of the Playground. (Actually, you should know that graphical pictures exist within a little transparent background square. It's the location of this frame that is "gotten" with GET LEFT/RIGHT/TOP/BOTTOM OBJECT. So when you use the selection tools to bring in handmade graphics, you'll want to draw the little square as close as possible to the edges of your picture so it doesn't carry around a big, transparent frame.) But what if one move causes that leftmost dot to leap over the boundary of the Playground, so that from one second to the next the C-bit goes from "less than", to "greater than," without ever actually hitting the "equals" spot?




Illus. 12B The object coordinate in the AC can switch from "less than" (the Playground edge) to "greater than" in one move.

Using JUMP >= allows the programmer to be prepared for this possibility, and to make sure the ball bounces back the other way, as planned, even if its edge hops over the edge of the Playground instead of actually "bouncing" off it.




Illus. 12C JUMP >= allows you to account for and control a range of possible outcomes from the COMPARE command.

It will be up to you, the programmer, to decide which jumping conditions are right for the programming task at hand.

## LET'S PROGRAM!

This assignment is based on the sample code we just finished studying, so *don't peek* back at the sample code until you've given this exercise a try!

1. Launch FM and open Program7.fmp At the FM Welcome window, click on Open Existing Programs. Navigate through c:\, Fundam, Manual, Practice and Program 7. Then open the program marked Program7.fmp.

   This program is basically the same as the one that appeared earlier in the chapter, only the ball doesn't bounce. You need to add the commands to fix that. For every time the ball moves, you'll have to do the following:

   Get all the edges of the object, each in turn.

   Compare their respective coordinates to the edges of the Playground.

   Jump to another marker to reverse the direction when any of the comparisons finds that part of the object is at a coordinate less than or equal to zero, or greater than or equal to 300. The labels and reversing code are already in place for you.

2. Use GET-LEFT OBJECT to get the leftmost coordinate of the ball object, into the AC.

3. Use COMPARE NUMBER to compare the coordinate in the AC with the horizontal coordinate corresponding to the left edge of the Playground.

4. Use JUMP <= to tell the computer to jump to the marker @sidewallbounce if the comparison finds that the left edge of the object is at a point less than or equal to the left edge of the Playground.

   Great! You've just finished your instructions for checking one of the four sides of the object!

5. Use LOAD BOX ballbox in order to get the ball object back into the AC for another check. (Can you see what would happen if you missed this step before trying to check the other edges of the object?)

6. Repeat the four steps above until all of the remaining three sides of the object have been checked.

7. Use the command JUMP ALWAYS @move when you've finished making all four of your checks, in order to make sure the ball keeps moving if it's *not* touching any of the sides.

8. EXTRA CHALLENGE: If you've got your ball bouncing and you're ready for some more fun, make this program into a pinball game. At the beginning of the code, construct one or more barriers to place somewhere in the Playground. Put each barrier you construct in its own box, with box names like, "barrier," "barrier2," etc. Then, under the first MOVE OBJECT command in the program, insert one TOUCHING OBJECT command for each barrier box. After each TOUCHING OBJECT barrier#, use JUMP = @sidewallbounce to get the ball going in the opposite direction *if* the ball happens to be touching that particular barrier. (If the ball isn't touching any of the barrier objects, then the computer will just drop down and execute the other COMPARES you already ordered for bouncing off the Playground edges.

## Programming Tips and Tricks

Don't forget to adjust your COMPARE NUMBER, and JUMP inputs to fit the edge of the Playground that you're currently checking.

Did you notice that we asked you to use LOAD BOX ball as the first new command, after MOVE OBJECT, even thought the ball is already in the AC at this point? It's good programming *style* to keep things parallel, and since the next three groups of similar commands had to start with LOAD BOX ball (since the command beforehand will leave a number in the AC), we're better off also making the first set of GET commands begin that way, as well. The code is easier to read and understand that way because it is consistent.

Go back and take a look at the way MULTIPLY NUMBER was used to manipulate animation in the sample program above. Can you think of a way to use it in this same way in any of your existing programs? Keep in mind the versatility of these NUMBER commands as you move into new programming projects. How can you use them to help you better manipulate different aspects of your programs?

## Ticks! (Another Fun Thing You Can Get)

Wait! (and I really do mean "wait," because that's what Ticks are all about). Don't run for the bug spray! I'm not talking about nasty bugs you pick up when you're hiking in deep bushes…I'm talking about the programming kind of Ticks, which are actually little teeny time particles…1/60 of a second, to be exact. They may seem like nasty little buggers at first, but once you get used to using them, they can actually be quite friendly, especially if you want to tell the computer to stop and take a deep breath for five seconds or so while the user catches up with what's going on.

(You've probably noticed by now that the average CPU is quite swift in doing your bidding. Instructions that took you half an hour to figure out and type in can take the computer half a minute to run through!)

### ✔GET TICKS

GET TICKS takes a reading off the computer's internal clock to find out the number of Ticks that have transpired since midnight. So at one second past midnight, a GET TICKS command would put a number 60 into the AC. If 60 ticks make one second, then 180 ticks make 3 seconds, 600 Ticks make 10 seconds, and so on. With that in mind, take a look at this sample code. It essentially takes the number

for *now*, calculates what *now* will be three seconds in the future, and then simply waits for the not-at-all-distant future to arrive.

```
    GET TICKS
    ADD NUMBER 600
    STORE BOX later
 @wait
    GET TICKS
    COMPARE NUMBER later
    JUMP >= @action!
    JUMP ALWAYS @wait
```

As long as we're comparing and jumping, try these new string tricks...


Now that you have variables and various jumps in your bag of tricks, look at these new string commands. COMPARE STRING works with strings in exactly the same way it works with numbers in COMPARE NUMBER: It compares what's in the AC with something else that you specify.

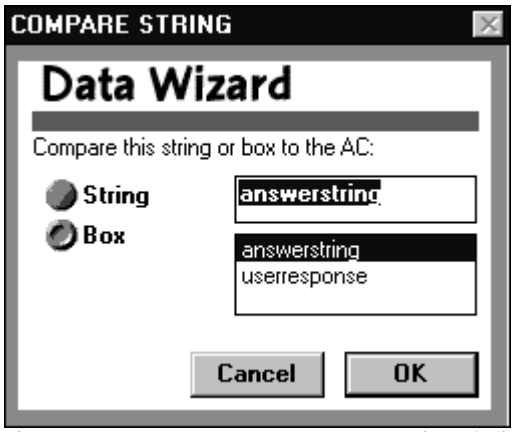



Fig. 12-1 COMPARE STRING Data Wizard dialog

Look at the program segment in the example below:

```
LOAD STRING "arctic gray fox"
STORE BOX answerstring

LOAD STRING "I'm thinking of an animal from the Antarctic…Can
you guess what it is?
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE BOX guesstring
```

```
COMPARE STRINGanswerstring
JUMP =@congratulations
JUMP <>@try again
```

GET-LENGTH STRING tells the computer to count the number of characters and spaces in a given string, and to put the resulting number into the AC.

DISSECT STRING is a little more complicated. It allows you to get at a particular character or group of characters from inside a string. When you use the command DISSECT STRING the Data Wizard will ask you for two number inputs. (Aha! So that's the connection to numbers!)

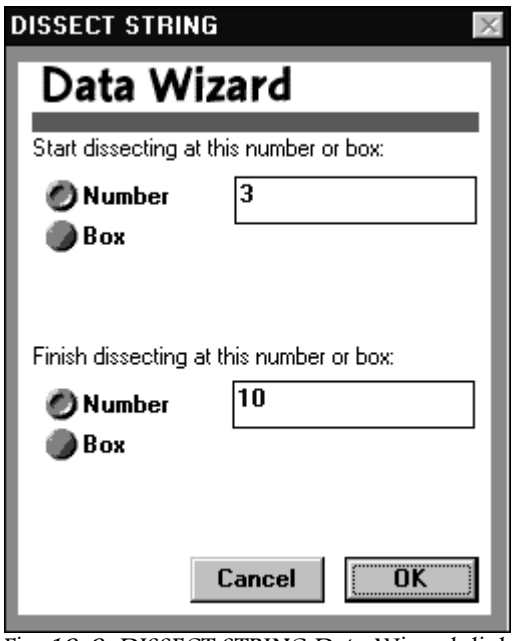


Fig. 12-2 DISSECT STRING Data Wizard dialog

The whole instruction will look something like this:

```
DISSECT STRING 3,10
```

That tells the computer to put in the AC all the characters (including spaces) from the third through the tenth in a given string.

So if the AC contains the string "I'm not sure this makes sense!" the command DISSECT STRING (3,10) pulls out "m not su" and puts that into the AC. DISSECT STRING is just another tool for setting up COMPARES and JUMPS in a program.

## "Issect-Day Ing-Stray!"

Neither GET-LENGTH STRING nor DISSECT STRING are very commonly needed for simple programs. But one programmer found a use for both of them in this program that will take any word a user types in and translate it into Pig Latin. That's more complicated than it sounds! Let's take a look.

```
@beginning
        LOAD STRING "Hello, give me a word to translate into Pig
        Latin.\"
        WRITE-SCREEN STRING
        READ-SCREEN STRING                          }gets and stores word
        STORE BOX englishWord
        LOAD NUMBER 1                               }gives #i.d. to 1st
                                                    letter
        STORE BOX currentLetter
@findfirstvowel
        LOAD BOX englishWord
        GET-LENGTH STRING
        STORE BOX stringLength                      gets string's length
        COMPARE NUMBER currentLetter                makes sure word is
                                                    long enough
        JUMP < @novowels                            jumps if it's not

        LOAD BOX englishWord                        loads user word
        DISSECT STRING currentLetter, currentLetter takes out current
                                                    letter
        UPPERCASE STRING
        COMPARE STRING "A"
        JUMP = @foundfirstvowel                     }compares current
                                                    letter to vowels
        COMPARE STRING "E"
        JUMP = @foundfirstvowel
        COMPARE STRING "I"
        JUMP = @foundfirstvowel
        COMPARE STRING "O"
        JUMP = @foundfirstvowel
        COMPARE STRING "U"
        JUMP = @foundfirstvowel

        LOAD BOX currentLetter
        ADD NUMBER 1                                }puts next letter in
                                                    box
        STORE BOX currentLetter
        JUMP ALWAYS @findfirstvowel
```

```
@foundfirstvowel
        LOAD BOX currentLetter                    } checks initial
                                                    letter,:(vowel?
                                                    consonant?)
        COMPARE NUMBER 1
        JUMP = @firstletterisavowel
        JUMP <> @firstletterisaconsonant

@firstletterisavowel
        LOAD BOX englishWord                      } translates words
                                                    with initial vowel
        APPEND STRING "way"
        STORE BOX piglatinWord
        JUMP ALWAYS @end

@firstletterisaconsonant
        LOAD BOX englishWord                      } translates words
                                                    with initial consonant
        DISSECT STRING currentLetter, stringLength
        STORE BOX startOfNewWord
        LOAD BOX currentLetter
        SUBTRACT NUMBER 1
        STORE BOX currentLetter
        LOAD BOX englishWord
        DISSECT STRING 1, currentLetter
        APPEND STRING "ay"
        PREPEND STRING startOfNewWord
        STORE BOX piglatinWord
        JUMP ALWAYS @end

 @novowels
        LOAD STRING "try a word with vowels"
        STORE BOX piglatinWord                    } asks user for
                                                    translatable word
                                                    (always a good idea to
                                                    anticipate weird input!)
        JUMP ALWAYS @end

 @end
        LOAD BOX piglatinWord
        LOWERCASE STRING
        WRITE-SCREEN STRING                       } prints Pig Latin.
                                                    word, or asks for
                                                    translatable word
        LOAD STRING "\"
        WRITE-SCREEN STRING
        JUMP ALWAYS @beginning

        EXIT PROGRAM
```

Whew! That's a lot of jumping around!! But I hope you took the time to really *dissect* this code. It's a fun program, and worth the

effort. (If you want to see it run, it's waiting in the Examples folder inside Fundam and Manual in your Existing Programs.)

## You Can Compare Objects Too!

✔COMPARE OBJECT

You can compare objects, too! COMPARE OBJECT works in a slightly different way than COMPARE STRING and COMPARE NUMBER. When the Data Wizard appears, it lets you pick either a box (which should contain an object inside it) or an object type or blueprint.

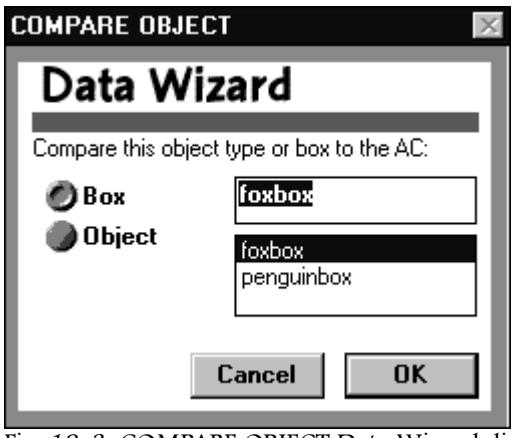



Fig. 12-3 COMPARE OBJECT Data Wizard dialog

If you pick a box, when the user makes the comparison, the two objects are "equal" if the object in the AC is equal to the object in the box. If you pick the name of an object blueprint, the comparison is "equal" if the object in the AC was made from the blueprint you selected.

Take a look at this sample code which makes up a matching game for young children. See if you can figure out what it does and how it makes use of COMPARE OBJECT. The first chunk of code sets up the game, and the second chunk is a sub-task which has been linked to all the program's objects as a click-task.

Setup:

```
CONSTRUCT OBJECT mouse
SHOW OBJECT
CONSTRUCT OBJECT snake
SHOW OBJECT
CONSTRUCT OBJECT turtle
SHOW OBJECT

LOAD STRING "Click on the animal which is a mammal."
WRITE-SCREEN STRING

END SUB-TASK
```

```
Click-task:
COMPARE OBJECT mouse
JUMP <>@wrong
LOAD STRING "That's correct!  Good job!\"
WRITE-SCREEN STRING
WAKE MAIN
JUMP ALWAYS@end

@wrong
LOAD STRING "Try again!\"
WRITE-SCREEN STRING

@end
END SUB-TASK
```

It's pretty easy to see how this program uses COMPARE OBJECT. There are three animals on the screen, and when the child clicks on one of them (which you remember will automatically transfer the clicked-upon object into the AC), the click-task simply compares the child's choice with the mouse object and jumps accordingly. (You can see this program in action by clicking Open Existing Programs at the Welcome window, and navigating through c:\Fundam\Manual\Examples\Demo2, and opening Demo2.fmp.

## It's Not as Simple as It Looks

Perhaps more difficult to understand in the above code are the commands END SUB-TASK, in the set-up portion, and WAKE MAIN in the click-task sub-task. Isn't the set-up the main task here? Actually, it's not. The main task for this program has three commands:

```
EXECUTE SUB-TASKsetup
SLEEP MAIN

END PROGRAM
```

Why not just put all the stuff in setup in the main task? Well there is a reason. In this case, if "setup" wasn't in a task by itself, then the program would have no way to EXIT. It would run on forever unless someone clicked the Stop button on the Toolbar. (Can you see why?) This may be functional, but it's *bad style.* So, in fact, this program is more complex than you'd think at first glance, because it presents this stopping problem. That's why the programmer decided to make one extra sub-task besides the one she used for her click-task. In the next chapter, we'll learn more about how sub-tasks can help you break such problems down, and solve them with style.

# CHAPTER 13
## Breaking Things Down with Sub-Tasks



Commands Introduced:

## ----- Teacher's Journal -----

Brian and Tove had a pretty complex program going. They had a caterpillar object with a click-task that made it inch across the screen, while two different birds swooped down and tried to eat it.

"If he makes it to the other side of the Playground," Tove explained, "we want the caterpillar to turn into a chrysalis and then a butterfly, and fly away..."

"And it would say something like 'You win! Butterflies rule!' in the Conversation window," Brian added.

"Cool!" I said.

"Yeah, but wait," Tove said, "If he doesn't make it across without getting dive-bombed, then obviously he can't turn into a butterfly…"

"AAAUUUUUUUUGGGGHHH! I'm DEAD!" Brian said, dramatizing the other possible outcome their game would hold.

"But it's too many JUMPS and everything, and we can't keep track of it all," Tove said, her shoulders sagging under the weight of all the possibilities.

"What you guys need are some sub-tasks," I said.

"We need things to be SIMPLER, not MORE COMPLICATED," Brian protested.

"Sub-tasks do make things simpler," I said. "You already use them all the time."

"We do?" Tove and Brian said together, giving each other a 'There-she-goes-again!' look.

"Sure. Tove, tell me what you did this morning before you came to school."

"Took a shower, ate breakfast, brushed my teeth, and made my lunch. So?"

"So, you just used sub-tasks to make things simpler."

"I did???"

"Yeah. You knew you couldn't really tell me EVERYTHING you did or it would take up the rest of the period and go into break time. So instead, you named the sub-tasks...."

"Ohhhhhhhhh," Tove said. And then she gave me a sheepish grin. "Okay, so you're not TOTALLY out of your mind..."

## Branching Out

Now that you're "branching out" in your programs, it's time to get back to the whole business of sub-tasks. You already know of one way to use sub-tasks, as sub-tasks for your objects. But that's just scratching the surface of what sub-tasks can do for you. Let's dig a little deeper.

Remember that the term "click-task" refers to any sub-task that has been connected to a particular object in the object designer, and that a sub-tasks is a separate section of program code written in its own Task window. Using sub-tasks as click-tasks is just one of the many ways to link them (or, in tech speak, "call them in") to your main program. Once they are written, sub-tasks can be used in a variety of ways.

## Click-Tasks and Key-Tasks

Sub-tasks can be used to bring the user into the action in two different ways, as click-tasks and as key-tasks. If you need to review how to make a new sub-task or how to link a sub-task to an object to create a click-task, refer back to chapter 8.

But let's say you want to make a program with a walking guy object. In the main task, you construct, show, and store him. But you don't want him to just automatically start walking; you want the user to be in control.

You could make a click-task, but it's sometimes better to have the user hit a key on the keyboard, especially if speed is of any importance. So let's say you want the Walker Guy to take one step for every time the user hits the spacebar on the keyboard. How do you think you'd accomplish this?

FUNdaMENTAL offers a really easy method for making key-tasks. Like a click-task, a key-task is a sub-task that gets called into play only when the user does something while the program is running. As the name "key-task" suggests, the computer will only execute this sub-task when the user strikes a designated key.

You can make a key-task out of any sub-task by simply clicking on the button marked with a picture of a key (the kind you use to open a locked door), which appears near the Use button in the sub-task window.


Fig. 13-1 The key-task button appears in the upper right-hand region of the sub-task window.

(Please note that the main Task window does not have this button. Can you explain why it would never work to make a click-task for a key-task out of the main task?)

Clicking on this Key button brings up a dialog box asking you to "assign" a keyboard key to this task.

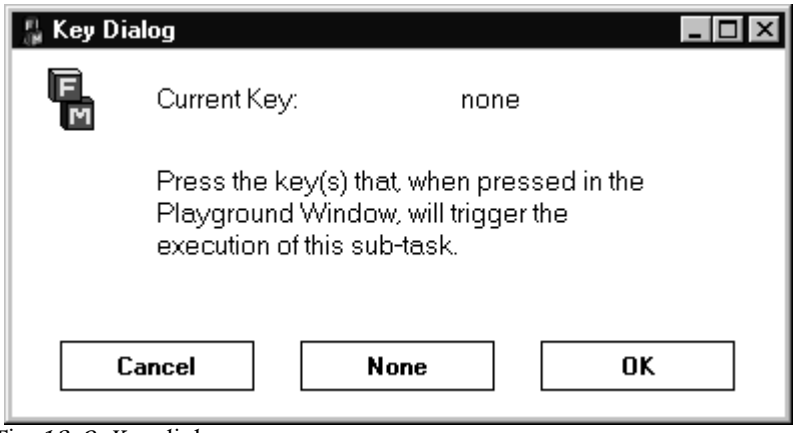



Fig. 13-2 Key dialog

All you have to do is press the key on the keyboard that you want to have linked with this sub-task. Doing so automatically creates a key-task so that the sub-task will run whenever the user presses the designated key while the program is running.

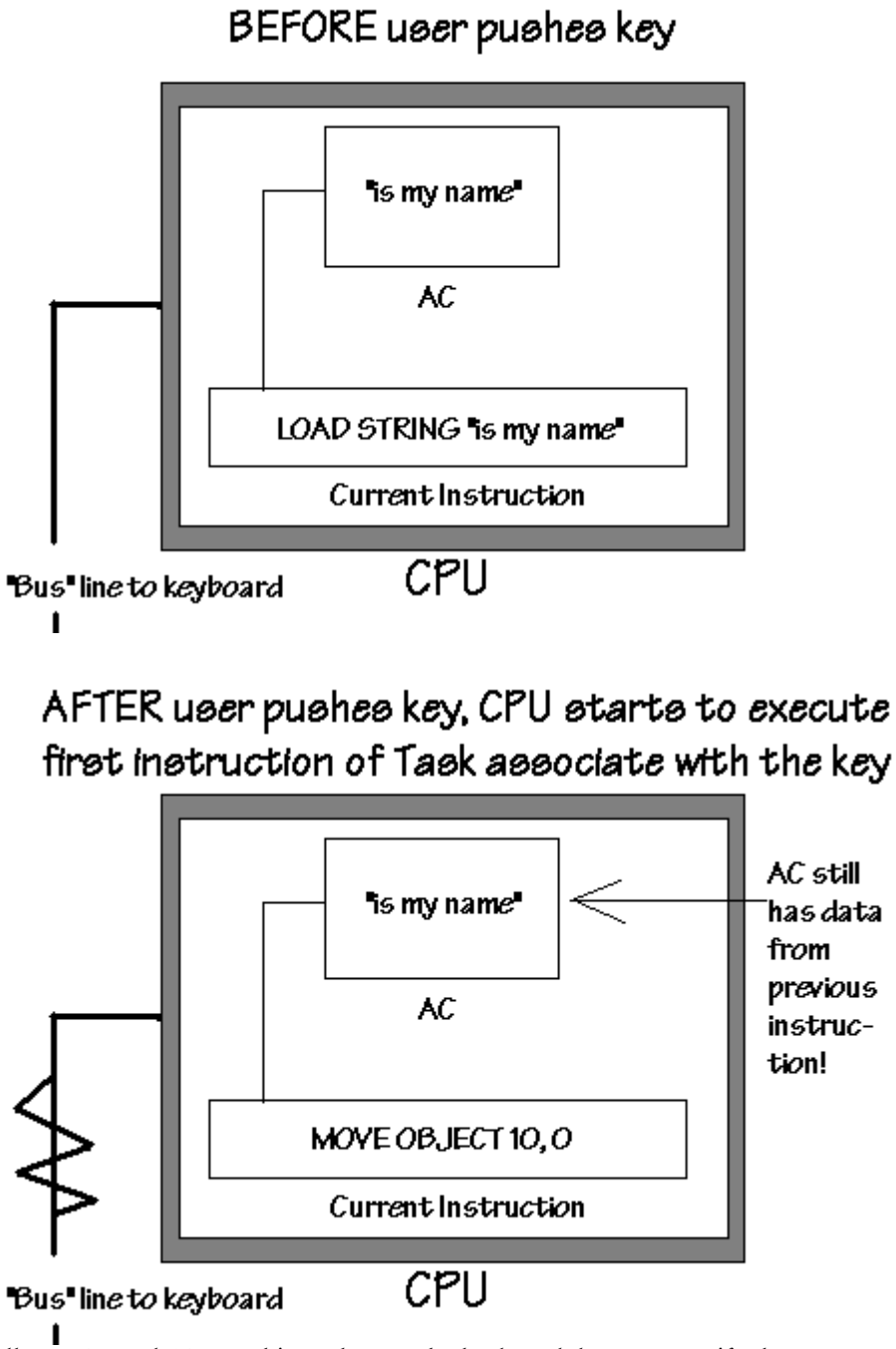
Key-tasks are different from click-tasks in one important way. Whereas you can begin a sub-task that you are using as a click-task with a command like MOVE OBJECT, you must always start a key-task associated with an object with CONSTRUCT OBJECT or LOAD BOX. Can you figure out why?

## What's Going On Inside the Computer?

When you use the mouse to click on an object, the computer automatically puts the clicked-upon object into the AC. Although a key-task is similarly associated with an object in your mind, pushing a key on the keyboard does not automatically put what you're thinking of into the AC. It simply causes the computer to start following whatever instructions are listed in the sub-task that you've linked with the key. If you start with the command MOVE OBJECT, the computer will either:

a) move whatever object is currently in the AC, which very well may be a different object than the one you have in mind, or

b) freak out entirely because there is not an object at all in the AC, but some other kind of data instead.




Illus. 13A and 13B Pushing a key on the keyboard does not specify the contents of the AC in the same way that clicking on an object does.

That's why you want to start your key-tasks with LOAD BOX, so that when the computer stops what it's doing and picks up this new list of instructions, the first thing it does is put the desired object into the AC.

## LET'S PROGRAM!

1. Return to Myprog1.fmp. You should already have at least one program with a click-task from back in Unit 1 when we took over the whole show and designed our own objects. For this assignment, you can return to that program, which should still be called Myprog1.fmp in your Fundam folder, or you can go back to any other program that you want to enhance with more tricks.

2. Select an object to be the recipient of a key-task. Make sure that the object you select is stored in a box, and make sure that the little globe symbol appears next to the box name in the list of boxes. (That way you'll know that you can load the object from a sub-task as well as from the main task. We'll be learning more about local and global boxes later in this chapter.) Now it's time to write the code for the sub-task that will determine what happens with the object when the user presses a key on the keyboard.

3. Create a new sub-task. Go to the Program window and click the New button for making a new sub-task.

4. Name the task something descriptive.

5. Start with the command LOAD BOX, and select the box for the object you've chosen for this key-task. (Unlike click-tasks, a key-task does not automatically put the desired object into the AC. You have to do that with a command.)

6. Write a really simple sub-task, no more than a few lines. Maybe this object could take a leap, or morph into a different thing, or disappear for a moment when the user touches the designated key.

7. Define this sub-task as a key-task. Look for the little button with a picture of a key on it. Click on it.

8. Select the key that will make it all happen All you have to do is press the key of your choice on the keyboard, and that key will become the trigger for your new sub-task. Click OK once you have selected the key.

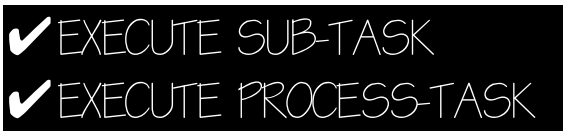
9. Test-run your program. While it's running, press the key you designated to see your key task executed. How did it turn out?

(Remember, before you share this program with another user, you should probably add a string or a sound that gives the tip-off about which key to press.)

## Sub-Tasks On Command

Clicking the mouse and pressing keys on the keyboard are not the only ways to call a sub-task into the action... in fact, among seasoned programmers (a category of folks that will very soon include you!) those are not even the most commonly used ways.

Another way to "call" sub-tasks is to use commands. Commands can call a sub-task either from within the main task or from another sub-task.




When you use either EXECUTE SUB-TASK or EXECUTE PROCESS TASK, the Data Wizard will ask you which of the existing sub-tasks you wish to call. You'll see the same list of sub-tasks that appears when you pull down the click-task menu in the Object Designer. It will be in the list that appears in the top portion of the dialog. (The two columns in the lower portion have to do with some special box tricks we'll be learning about in the next chapter. You can disregard them for now.)

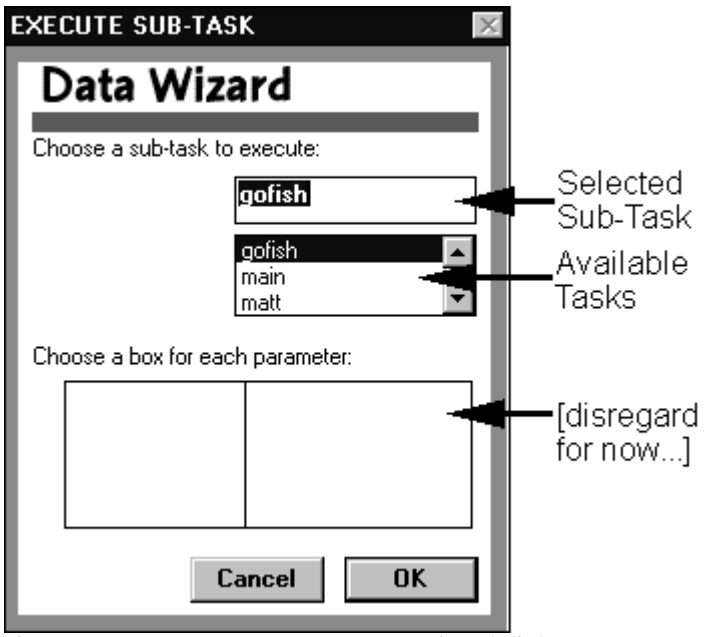



Fig. 13-3 EXECUTE SUB-TASK Data Wizard dialog

Select the sub-task you have in mind by double-clicking on it, and the EXECUTE command will automatically appear in the Instruction list followed by the name of the sub-task it's calling, like this:

EXECUTE SUB-TASK gofish

or

EXECUTE PROCESS TASK gofish

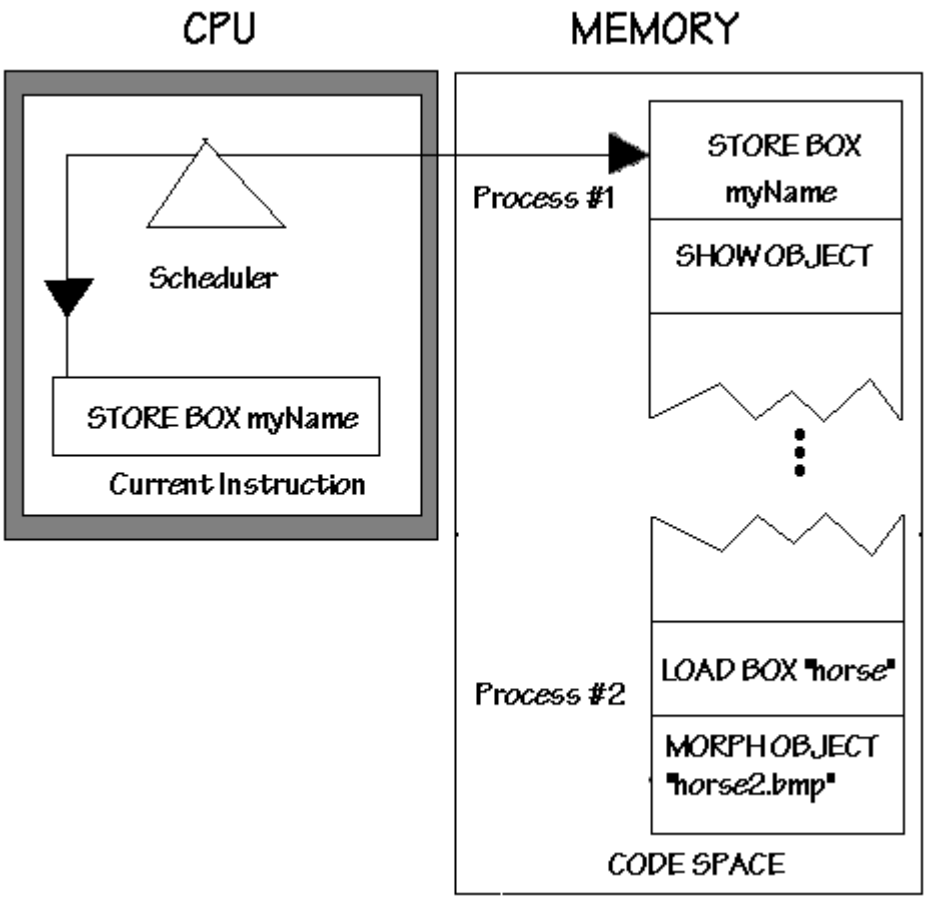
The command EXECUTE SUB-TASK is pretty self-explanatory. You're simply telling the computer to stop whatever it's doing in the current list of instructions, go over and carry out the commands in another list, and then return to the first task to do whatever comes after the EXECUTE SUB-TASK command. You're essentially telling the computer, "Go do some other things, and then come back and finish this stuff."

On the other hand, EXECUTE PROCESS TASK is more like telling the computer to flip a switch that starts another set of things happening, and then continue on with the original task, while the new task goes along as a separate process. Here, your message to the computer is, "Do two things at once, at least until one of them's finished."

## What's Going On Inside the Computer?

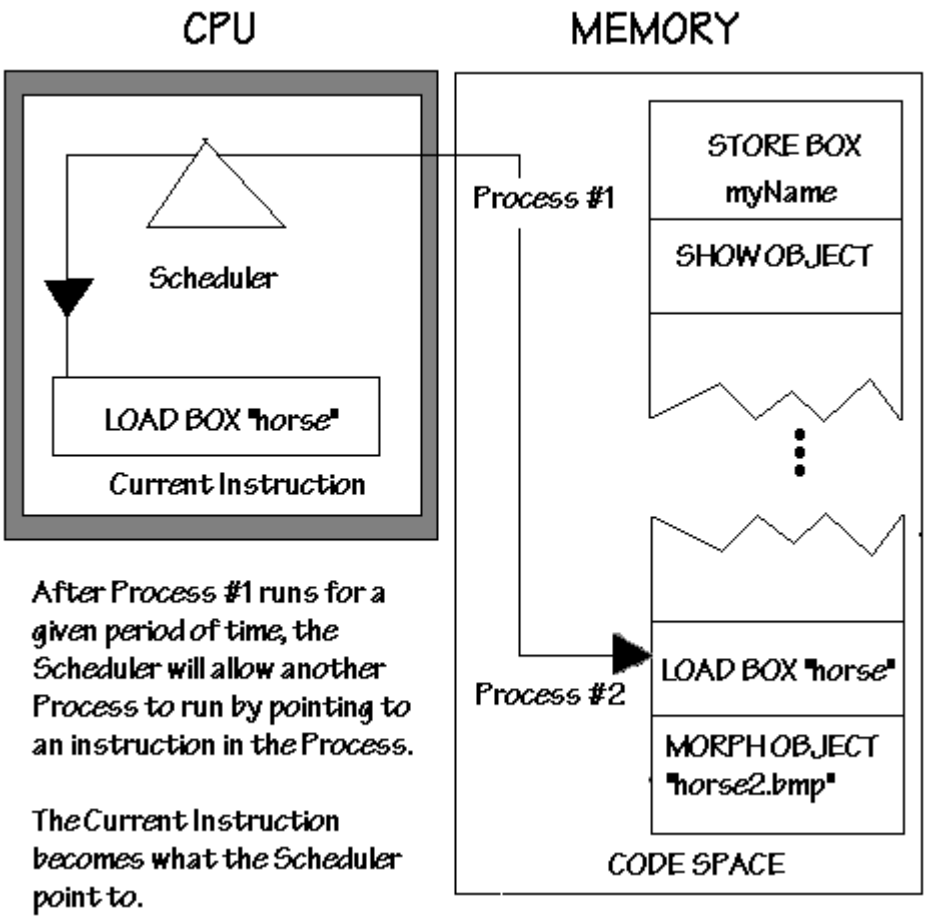
When the computer is in charge of accomplishing two different tasks at one time, it is said to be running two separate "processes." Since the computer can only do one thing at a time, this may seem impossible. In fact, it is. The computer is never doing more than one thing at a time. When it's in charge of two separate processes, what it actually does is switch back and forth between them. It does a little of the first and then a little of the second, and then a little of the first again, and so on, until something gets finished. Luckily for us, the computer does everything really, really fast. So even though it's not really doing two, or three, or ten things at once, it sure looks like it is when the program is run.

When more than one process has been established in a program, a little thing called the Scheduler jumps into the action.




Illus. 13C The scheduler makes sure all processes get a fair shake.

The Scheduler is what makes sure each process gets its fair share of the computer's attention. Unless the computer gets instructions from the current process to move on to another, the Scheduler will intervene after a short amount of time and make sure the computer moves on to the next process in the lineup.




Illus. 13D The scheduler alternates between processes.

## "What Can Sub-Tasks Do for Me?"

Like the two SOUND commands (PLAY SOUND and PLAY-N-WAIT SOUND), the two sub-task commands will offer distinct advantages to your program. The choice you make will depend upon what best accomplishes the effect you're trying to achieve when your program is run.

It's clear that click-tasksk, and key-tasks are useful to make your programs interactive. But what can be gained by using these other kinds of sub-tasks?

Sub-tasks, in general, enable you to break down complex programs into component parts so you can accomplish complicated tasks with style and efficiency. As you continue to advance in your programming, try to keep your mind open to different ways you can break things down into smaller chunks and try dealing with them separately in sub-tasks.

Process-tasks, in particular, are useful for achieving flexibility in your program. They're the ones that really make the computer appear to do several things simultaneously. You can set as many of

them spinning as you want to, and get a lively, circuslike atmosphere in your animation programs.

Or you can use them to keep a running set of instructions or soundeffects going in the background together with the main action.

## Everyday Sub-Tasks

As prickly as they may sound in the context of programming, sub-tasks are really old friends of yours. You use them all the time, as Tove only reluctantly admitted. ("Do not!" "Do too!")

Let's say you have a busy day ahead. You need to deposit money at the bank, go to the grocery store, cook supper, learn a little bit about programming, and plan tomorrow's lesson for your students. Each of these items on your "to do" list is made up of a whole bunch of smaller "commands" that you give to yourself as you complete the task.

For example, Deposit Money at the Bank breaks down into a lot of different little things once you're standing in front of the ATM – things like: take out your wallet; open your wallet; take out the check; pick up a pen; sign your name; etc.

But when the alarm goes off in the morning, you'd never make it out of bed if you tried to actually tally up everything you'll be doing that day. Only when it comes time to EXECUTE SUB-TASK deposit, do you deal with all the smaller details. When you think about it this way, it's easy to see how sub-tasks themselves can break down into sub-tasks. Inside the Deposit sub-task, for example, you might find EXECUTE SUB-TASK lookupaccount#, or EXECUTE SUB-TASK signature.

## Decomposition and Modularity

Now we're getting into the nitty gritty! My programmer buddies have a name for all this breaking down of stuff: "decomposition." Aside from making it easier to face your day–or your program– decomposition also allows you to reuse certain sub-tasks. For example, you EXECUTE SUB-TASK signature every time you write a check, send a letter, or pass the guest book at a wedding. This kind of reusable sub-task is said to have a high degree of "modularity."

The more you practice with programming, the more you'll find ways to create self-contained, or modular, sub-tasks that you can reuse in many different programs. In fact, there's a FUNdaMENTAL interface feature which allow you to do just that. It's called the Task Importer. It looks like this:

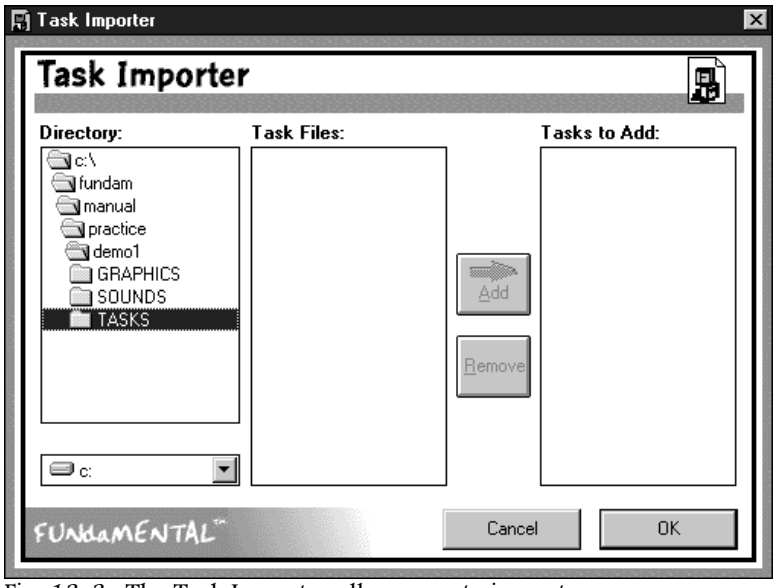



Fig. 13-3a The Task Importer allows you to import whole sub-tasks from other programs

You get to it by clicking the Add button in the Program window. Once it's open, you use it basically the same way you use the Graphics and Sound Importers. Navigate through the directories to find the program that has the sub-task you want to reuse elsewhere. Open the folder marked "tasks", and then highlight the task you want and click on the Add button in the Task Importer. The task name will now appear in the right-hand column of the Task Importer. Close it up when you've grabbed all the reusable tasks that you need.

## Remember Ticks?

They're the little critters that allow you to control time lapses in your programs. A "wait!" function with ticks is a perfect miniprogram to separate into its own sub-task, as in the example below.

```
    GET TICKS
    ADD NUMBER numseconds
    STORE BOX later
@wait
    GET TICKS
    COMPARE NUMBER later
    JUMP >= @action!
    JUMP ALWAYS @wait

    END sub-task
```

To make it truly recyclable, you want to put a box/variable, instead of a number constant, with the ADD NUMBER command (something like ADD NUMBER numseconds). That way, you can use whatever number of seconds best serves your "waiting" needs.

## Programming with Sub-Tasks

This wait function is just one example of how you can take care of a portion of your program's business in a separate sub-task. In this same way, you can move objects, print instructions in the Conversation window, or play sounds. There's no single correct way to achieve good decomposition no formula for perfect modularity. This is the true art of programming. Your own style will grow with experience. (But just remember: Sub-tasking is a natural thinking process in which you engage every day of your life, from morning 'til night!)

So if you are trying to do something in your main task that seems to break down into several, different functions, or if you find yourself writing miles of code in a single task, ask yourself if you could find a place for all or part of your plan in a sub-task.

## Whose Box Is It Anyway?

Let's say you're working in a sub-task and you want to use the command LOAD BOX. You're already starting to think like an expert (and we're not even to unit 3 yet!), and so suddenly you stop and ask yourself: "Wait a minute! If I stored a box in the main task, can I load it into a sub-task?"

Hey! Good question! (Aren't you glad you thought of it?)

And the answer is...yes, and no. Two answers, because there are two kinds of boxes: global boxes and local boxes. In using the command STORE BOX, you may have noticed that when you created a box, it added the new box name to the list on the right side of the Task window. And next to each box in this list, there is a picture of a little globe. If you click on the globe, you'll see that it changes to a picture of a little house.

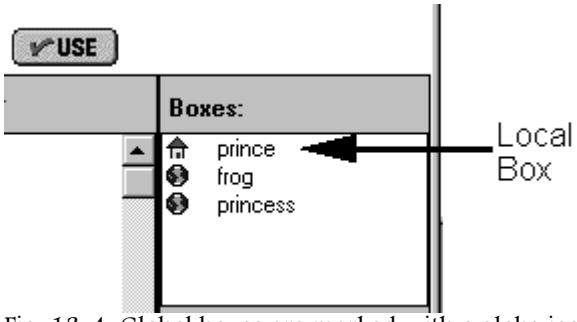



Fig. 13-4 Global boxes are marked with a globe icon, and local boxes are marked with a house icon.

That means that the box is now "local," or specific to the sub-task in which it appears. Local boxes will only appear in the box list for one task in the program. The boxes next to the globe, on the other hand, are "global" and will appear in the box lists for all the tasks in your program.

So, let's start with global boxes, the kind that go with the "yes" answer. (Can you even remember the question? Here it is again: Once a box is stored away in one task, do all the other tasks in the program have access to that same box?) If a box is designated as a global box, then that means that it resides in a storage area that is shared by all the tasks in your program. Every task is allowed to "walk right in" and get boxes out.

On the other hand, local boxes come from a private storage bin belonging to one specific task. Sometimes local boxes get filled up with data that's "passed in" from another task. But for now, we'll focus on local boxes that have to be stored (or filled up) within the task that uses them, and labeled as local. In this case, whenever the computer is following the instructions of that sub-task and needs to load a local box, it dips into the special, private bin, rather than going to the main storage area.

## "Local" Benefits

At first, I couldn't see why all the programmers around me were insisting that local boxes were the way to go. Hey, I reasoned, why not just store away everything once, and let all the tasks share. Sharing is good, isn't it?

### Organization and Readability

Not in this case. Not usually, anyway. The main reason to use locals whenever possible is to keep things organized, which, in programming, translates into readability, a key component of good programming style. A fairly complex program may use dozens of boxes altogether. When you're working in a particular sub-task, you will see a box list that contains all the program's global boxes, plus only the local boxes that pertain to that particular sub-task.

All the other local boxes that go with different sub-tasks are hidden. That way, it's a lot easier to sift through the list and find the box you had in mind. Using all global boxes would be a little like storing duplicates of everything you use in your actual household–from your tooth brush to the dog's collar to your frying pans–in every single cabinet in every single room. (Yikes! What a mess!)

### Modularity

The other main reason to use local boxes applies to more advanced programming, but it's a good idea to start thinking along these lines early in your programming career. It has to do with the value of modularity, that is, trying to make sub-tasks that can be used in more than one program. That may be hard to imagine now, but it's a big part of the real stuff of programming. Imagine, for example, that you had a sub-task that used the command LOAD BOX bunnyhop for a global box containing a hopping distance that had been stored in the program's main task. Suppose you then tried to use that sub-task

in another program called Rabbit Race. During that program's "syntax check" (when the computer runs down your list to make sure all the necessary elements are there), the computer would look for the definition of the "bunnyhop" box, which, of course, wouldn't be there because it would be stranded back inside the Easter Bunny program as a global variable–CRASH!

### Moral:

When it comes to boxes in sub-tasks, forget everything your teacher told you about sharing, and use local boxes whenever you can.

## LET'S PROGRAM!

1. Launch FM and open Program6.fmp. At the Welcome window, click on Open Existing Programs, then navigate through c:\, Fundam, Manual, Practice, and Program 6, and open Program6.fmp. Once this program appears in the Files window, double-click to open it. (It's the frog prince; remember him?) We're going to change this program so that it includes EXECUTE SUB-TASK and EXECUTE PROCESS-TASK.

2. Review the code to find the chunk you'll use as a sub-task. Find the part that makes the frog turn into a prince and walk into the sunset after the kiss.

3. Use the cut function in the Edit menu to remove these lines of code from the main task. (Make sure you leave the "@kiss" marker in place, though; you're still going to need it. ) The copy, cut, and paste functions work the same way they do in conventional word processing applications. When you use cut or copy, the selected text is either removed or copied to the computer's clipboard. The paste function will paste the last thing cut or copied into the chosen location. To select more than one instruction to be cut or copied, click on the line at the top of the sequence and drag the mouse down until all the desired lines are highlighted.

4. Go to the Program window and make a new sub-task and name it "change!" Use the Window menu on the Toolbar to select Program window. Click on the New button and type the name of the sub-task into the Data Wizard dialog that appears.

5. Paste the code you cut from the main task into this sub-task. Highlight the first line of the Instruction list in the new sub-task window, and then select Paste from the Edit menu on the Toolbar.

6. Go back to your main task. (You can do this by clicking directly on it if you can see any part of it, or by finding the Program window and double-clicking on "main" in the list of tasks in your program.)

7. Add the instruction EXECUTE SUB-TASK change! It belongs underneath the "@kiss" marker. Your program should run exactly the same as it did when this stretch of code was part of the main task list.

8. Go back to the Program window, and click on the New button to make another new sub-task. Call it "madwitch.fmt" (You'll find witch graphics in the Graphics Library for this program.) Construct a witch object. Place and Show it. Then use the necessary MOVE and MORPH commands to make her have a tantrum in the background as her spell is foiled by the frog's kiss.

9. Go back to your main task, and review the code to find a good insertion point. Find the place in the main program where you want the mad witch stuff in the new sub-task to start happening.

10. Insert the instruction EXECUTE PROCESS TASKmadwitch and attach your new witch object sub-task to this command. Run the program, and see how it looks.

## Programming Tips and Tricks

Can you see the advantages of moving these small chunks of code into their own sub-tasks? What if you had a whole bunch of things going on in the program with several different possible outcomes, depending upon what was touching what? It's much easier to keep track of all these outcomes if they're in separate little tasks, all on their own.

A programmer friend of mine says that if he finds himself using more than 15 or 20 lines of code in one task, he figures it's time to stop and look for a way to break things down. Most times, he can find one! The Pig Latin program in the previous chapter is a perfect candidate for sub-tasks. Can you see how you'd break it up into separate sets of commands to stash in sub-tasks? Give it a try!

And remember the different advantages offered by the two commands EXECUTE SUB-TASK and EXECUTE PROCESS-TASK. With EXECUTE SUB-TASK, the computer will stop in its tracks upon reaching that instruction, and go execute all the instructions in the sub-task before picking up where it left off. With EXECUTE PROCESS-TASK, the computer begins executing the instructions in the sub-task as a separate process and so appears to be doing two things simultaneously.

Please note that the sample solution to this assignment can be found under the name Progrm6a.fmp in the Examples folder. (Fundam/Manual/Examples/Program6a)

# UNIT 2
## Highlights

### COMMANDS

LOAD STRING
WRITE-SCREEN STRING
READ-SCREEN STRING
PREPEND STRING
APPEND STRING
UPPERCASE STRING
LOWERCASE STRING
DISSECT STRING
PLACE CONVERSATION
RESIZE CONVERSATION
RESIZE PLAYGROUND
LOAD NUMBER
WRITE-SCREEN NUMBER
READ -SCREEN NUMBER
ADD NUMBER
SUBTRACT NUMBER
MULTIPLY NUMBER
DIVIDE NUMBER
REMAINDER NUMBER
RANDOM NUMBER
COMPARE NUMBER
TOUCHING OBJECT
JUMP =
COMPARE OBJECT
GET-BOTTOM OBJECT
GET-TOP OBJECT
GET RIGHT OBJECT
GET-LEFT OBJECT
GET TICKS
GET-LENGTH STRING
JUMP </>/<=/>=/<>
COMPARE STRING
EXECUTE SUB-TASK
EXECUTE PROCESS-TASK

### INSIDE THE COMPUTER

C-bit
Scheduler
Processes

# UNIT 3
## THINKING LIKE AN EXPERT

Now that you've had a good foundation in the basics of programming, it's time to begin exploring the way that expert programmers think. This is where the programming process becomes a true art. We'll be looking at advanced uses for variables, and new, dynamic methods for storing data, all with an eye toward creating programs that show "good style" with respect to readability and efficiency.

In Chapter 14 we'll revisit the central concept of boxes in FUNdaMENTAL, reviewing what we've learned so far, and adding some new dimensions to make the use of variables an even more powerful tool for achieving your program goals.

In Chapter 15 we'll look at a new way for storing large amounts of data in a way that is both flexible and orderly. We'll see how storing things in "gaggles" allows for more dynamic and elegant programming.

In Chapter 16 we'll learn commands that allow you to stash string and number data in separate text files. We'll see how this is useful for hanging onto data permanently. Even more fun, we'll see how structuring files well can allow you to write stylish programs that work like little machines, independent of the particular data they're handling.

In Chapter 17 we'll take a look at a new kind of data called "booleans," and use FUNdaMENTAL to have some fun with formal logic.

In Chapter 18 we'll take a focused look at the theme of good style. We'll revisit a variety of style elements that we've been developing throughout the manual, including decomposition, modularity, readability, and naming conventions.

# CHAPTER 14
## A Whole Little Chapter About Boxes

## ----- Teacher's Journal -----

"Okay!" I bellowed. "Company meeting, everyone!" I blew my wooden train whistle, and, slowly, clusters of kids began to leave their computers to come to the open spot in the middle of the room where we met to learn new commands and share ideas for programs and troubleshooting.

"What do we gotta meet about now?" Stephen groaned. He hated more than anyone to give up his precious work time.

"We have to talk about boxes," I began.

"BOXES?? AGAIN???" the class chimed in unison.

"Oh, yes!" I said. "Boxes again, and boxes still. Boxes are a big, BIG deal in programming.

# Boxes? Again?

Yes, boxes again, and boxes still, because having the right idea about making storage space for your data is one of the main skills you need to bring your programming up to a whole new level. Throughout the manual, boxes have resurfaced several times, each time with a new twist. Let's recap everything we've learned, and then add some new ideas so we can get a sense of how experts see boxes.

# Boxes in Review...

## Holding Onto Constants

When we first started using boxes, way back in unit1 we used them as a way for the computer to "keep track" of a particular object so we could get back to it later, once we had loaded something new into the AC. We focused on Red Ridinghood and he Wolf as we loaded them by turns into the AC to move them through the forest scene.

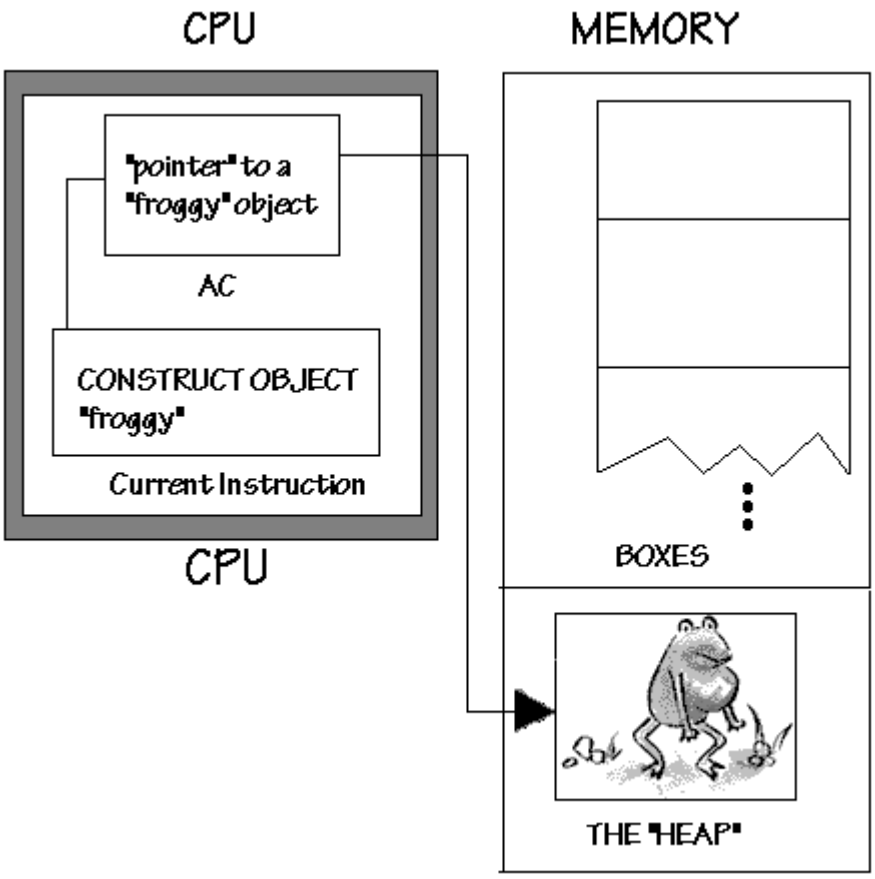
## Boxes as Variables...

Then, in unit 2, we saw how boxes themselves are actually more important than what goes in them. To the expert programmer, a box is really a variable, a nice open place that you can create within your program for storing different instances of a certain data type.

Instead of thinking, "I need a box for this #3," you think, "Here's where I'll put a numbox so that numbers–like, say, #3–can slip in and do their thing!" Expert programmers usually start the programming process by identifying all the variables they'll need to build into their program.

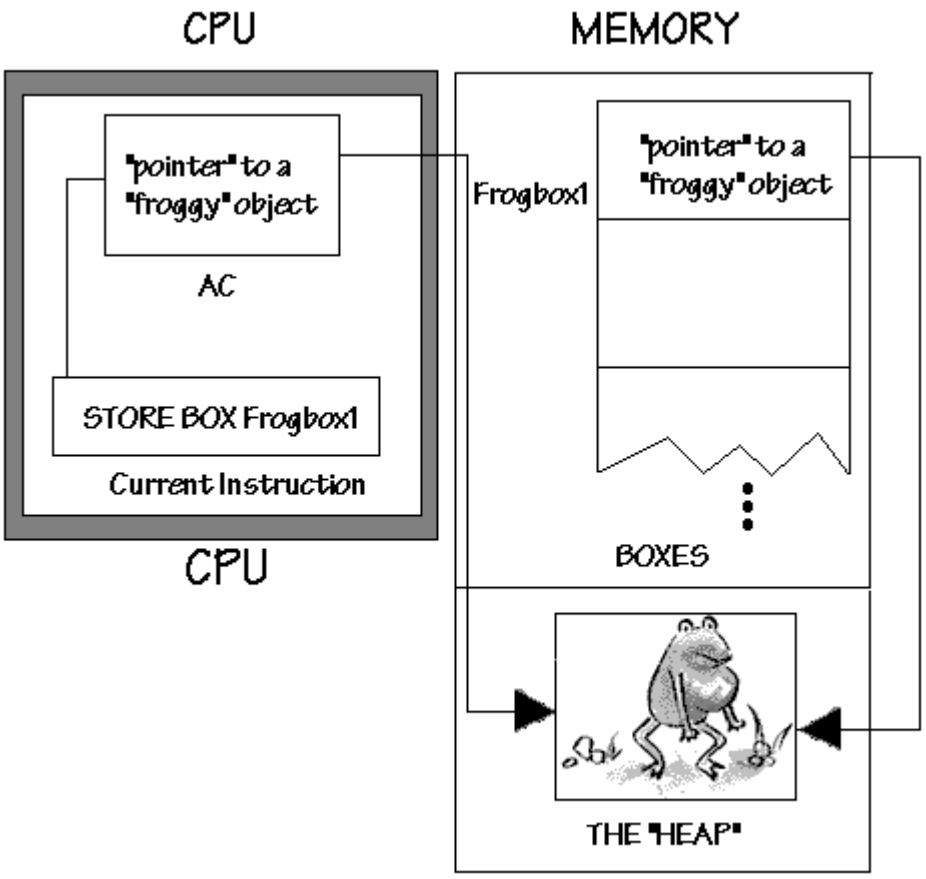
## Boxing Rules for Different Data Types

In unit 2, we also learned that different types of data use boxes differently. Here's the whole story. The command CONSTRUCT OBJECT actually sets aside a little chunk of memory in a separate place called the "Heap," and then constructs the object there.



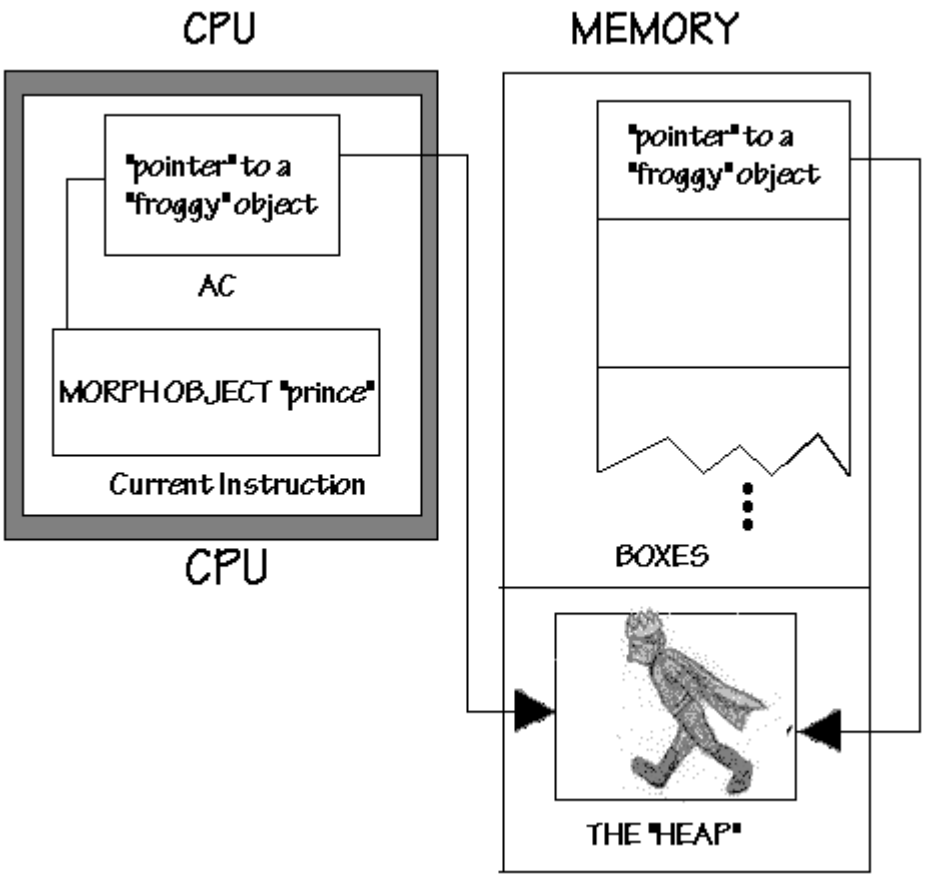

Illus. 14A objects are constructed in a portion of the computer's memory known as the "Heap."

When we store an object in a box, what we are actually storing is not a copy of the object itself, but rather a "pointer," or reference (like a "This-Way-to-objectfroggy" road sign, or a library call number).




Illus. 14B A pointer to the object is copied from the AC into a box.

When we later load box in order to manipulate the object, the pointer inside the box directs the computer straight to the original copy which resides in the Heap.




Illus. 14C After an object in the heap has been changed through a pointer in the AC, the pointer in the box leads to the same, updated copy.

Even if a constructed object is never stored in a box, it remains in the Heap, but the program will have no way to get back to it.

Strings and numbers are different. When we load a string or a number into the AC, there is no special memory space set aside elsewhere for the data. It exists only in the AC, and will be gone completely if it's bumped out of the AC before being stored. When we store a number or a string in a box, we are storing away an *actual copy* of this data. It's kind of like making a virtual rubber stamp to preserve the data's current form. The command LOAD BOX numberbox causes a copy of what's in the box to get "stamped" back into the AC. That's why, if we in any way change the number after loading it back into the AC, we need to use STORE BOX numberbox to update the contents of the box; otherwise it will still contain a copy of the original, unchanged form of the data.

## Local and Global Boxes

In the chapter on sub-tasks, we saw how programmers distinguish between boxes that are global and ones that are local. Although all programs will have some global boxes, which are accessible (or loadable) from within all the separate sub-tasks in the program, it's

best, whenever possible, to use local boxes, which are only accessible from the sub-task that contains them.

## And Now for Something New ... (How about a Little Variety?)

Okay, so you have a "Hop!" sub-task that makes your frog hop across the screen. He hops 30 times to the right every time you call this sub-task.

The code looks like this:

```
LOAD BOX frog
MORPH OBJECT longfrog
MOVE OBJECT (30,0)
MORPH OBJECT shortfrog
```

What a boring frog. You would like him to be able to hop a distance of 5 sometimes, and 50 other times, and every once in a while you want your frog to hop 63. But each time he hops, he goes no more nor less than 30.

Of course, you could write separate sub-tasks called HOP5, HOP50, and HOP63, but that's a lot of extra work, and you still couldn't account for every possible hopping distance.

But what if you could make one sub-task where you could plug in a different hopping distance each time it was executed? Well, I have good news... you can!

## Special Delivery:

### Received Boxes and Passed-In Parameters

You can send data into a box in a sub-task, which you can then use throughout the sub-task, by "passing in" the data as a parameter. Then the data is automatically stored in the received box and can be used throughout the sub-task.

To do this, the sub-task must have a "received" (or empty) box that can "receive" the information that is "passed-in" in the form of a parameter. When you execute the sub-task, you specify the parameters, or information that you are passing into the sub-task, and it will be automatically stored in the received box. You can send a number, or a string, or even another box, to be stored in a sub-task's received box.

## Defining Received Boxes

You can make a received box for a sub-task by using the area marked "received" under the regular box region in the task window for that sub-task.

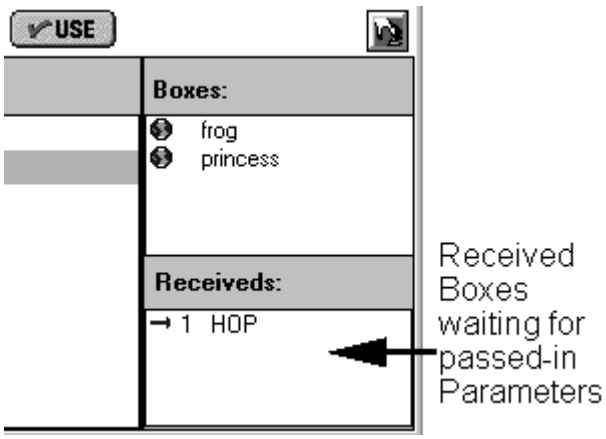



**Fig. 14-1 received box region in the sub-task window.**

The boxes in this list are empty until the computer follows the instruction to execute the sub-task, and brings in the specified parameters.

## Specifying Parameters

Now it's time to learn how to specify the parameters you want filled in. (Once you do, you'll notice the little blue arrow that appears next to each received box, which signifies that something is due to come *in*.) You do this in the EXECUTE SUB-TASK Data Wizard dialog. As soon as you select a sub-task in the list of available tasks, the Data Wizard will display a list of the empty, received boxes that need parameters in the column to the left of the lower portion of the dialog. Simply double-click in the adjacent space in the column to the right, and fill in the desired number, string, or box name. Now you have specified parameters!

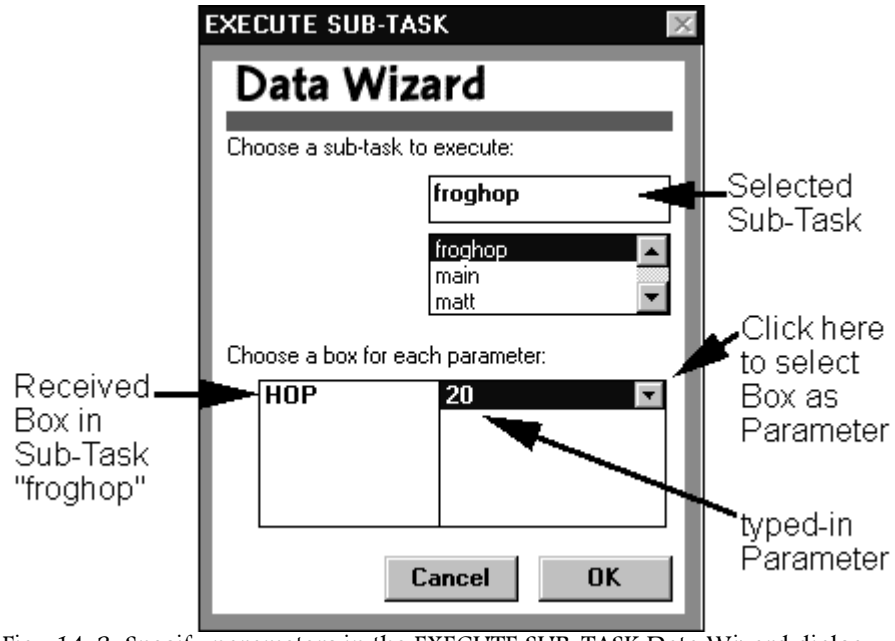



Fig. 14-2 Specify parameters in the EXECUTE SUB-TASK Data Wizard dialog.

## Parameters at Work

Let's go back to the hopping frog. We will change our sub-task "hop" so that it has a received box called "AmountToHop." Let's look at how our code changes, now that we are using the received box.

```
LOAD BOX frog
MORPH OBJECT longfrog
MOVE OBJECT (AmountToHop, 0)
MORPH OBJECT shortfrog
```

Then, when we execute the sub-task hop, we can "pass in" a number for the distance that we want the frog to hop. We could even pass in a box containing a number that we want the frog to hop. Here is what code would look like to make the frog hop 10, 54, 32, 21, and a distance the user specifies that is stored in a box called "userhop."

```
EXECUTE SUB-TASK HOP (10)
EXECUTE SUB-TASK HOP (54)
EXECUTE SUB-TASK HOP (32)
EXECUTE SUB-TASK HOP (21)
EXECUTE SUB-TASK (userhop)
```

Now you can have a sub-task that will work for any frog object and any hopping distance.

# LET'S PROGRAM!

1. Go back to Program2.fmp since we've been talking of hopping frogs. From the FM Welcome window, select, Open Existing Programs, and then navigate through c:\, Fundam, Manual, Practice, Program 2, and open Program2.fmp. Open this program.

   You should already have the MOVE and MORPH code there that you wrote while you were learning those commands in chapter 2. You can leave it there and work around it, or delete it, if you find it distracting. But make sure to leave the commands that construct, place, and show a frog object. We'll be needing those.

2. Store the frog object in a global box.

3. Create a new sub-task named "froghop." Do this by finding the Program window in the Window menu on the Toolbar (or by clicking on any part of the Program window that may be visible) and clicking on the New button. Name the sub-task in the Data Wizard dialog that comes up.

4. Define a received box named "hopamount." Click in the received box region directly under the regular box list in the Sub-task window, and type the box name in the highlighted space. Press the Enter key to enter it into the list as a defined received box.

5. Write a short sub-task to make your frog object hop. Use the box, hopamount, as the MOVE input for the x-direction. You really only need four commands in this sub-task. Load the global box containing the frog object, morph him to the stretched-out position, move him, and morph him back to the scrunched-up position. (visit the Graphics Library if you need a reminder of which graphic goes with which .bmp file name.)

6. Go back to the main task window. In the Window menu on the Toolbar, select main from the bottom of the list, or double-click on Main in the Program window.

7. Use EXECUTE SUB-TASKfroghop. Find the place in the code where you want the frog to start hopping and do it with your ready-made sub-task, instead of with MOVE and MORPH commands written directly into the main task.

8. Specify parameters for the sub-task's received box, "hopamount." When the EXECUTE SUB-TASK Data Wizard comes up, click on the sub-task "froghop" where it appears in the list at the top of the dialog. In the lower portion, the box name "hopamount" will appear automatically in the column to the left. Double-click in the right-hand column opposite the box name and type in any number in the highlighted space.

9. Use EXECUTE SUB-TASK in the main task several more times, and each time specify a different parameter to pass into the received box "hopamount."

## Programming Tips and Tricks:

Don't forget that you can use another box as the parameter that you pass into a received box. Just click on the little pop-down icon in the parameters column of the EXECUTE SUB-TASK Data Wizard dialog, and you'll see all the available boxes. Stop and think for a moment about the added flexibility this gives you.

Can you change your frog program so that a user can be in control of the parameter that gets passed into "hopamount"?

Or how about making "hopamount" receive a random number each time the sub-task is executed? (If you used RANDOM NUMBER and set a loop, you'd have a great way to execute the sub-task several times with a different parameter in "hopamount" for each round.

## One-Way Versus Two-Way Delivery

There's one more thing you should know about received boxes. They come in two varieties: one-way (or "in"), and two-way (or "in-out"). This has to do with storing rules again. With the one-way variety of received boxes, the data that was delivered to the box is not updated if the data is altered in the sub-task. With the two-way variety, it is. (This only applies, of course, to "actual value" data, like strings and numbers, since objects are not copied but only referenced in boxes.) Let's look at some examples.

Let's say you EXECUTE SUB-TASK ADD (numbox), and the sub-task gets a 5 passed in from "numbox" to a received box called "subnum." Even after the sub-task does its adding thing and changes its own copy of the number 5 to something bigger, the copy in the original "numbox" box stays the same. The received box in the "Hop!" sub-task is an example of an "in," or one-way received box.

On the other hand, "in-out" received boxes pass whatever they have back out, when the sub-task is finished. For "in-out" received boxes, the data from the passed-in box is actually changed if the received box is changed in the sub-task. So if you pass in a "numbox" that contains 5, and the "add" sub-task adds 5 to whatever's in "numbox," then the original "numbox" will now contain 10. This is something to consider, with respect to your program goals, when you decide which kind of received box you want to use. To specify an "in-out" received box, simply click on the little blue arrow that

appears next to the box name in the “received” box list in the sub-task window. The icon will now show a little red arrow pointing *out* underneath the original blue arrow pointing *in*. To return a received box to its original, one-way status, just click again on the same place and the double arrows will revert back to a single, blue arrow.

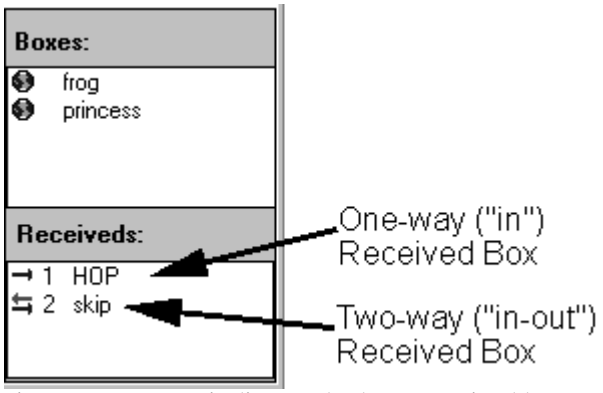



Fig. 14-3 Arrows indicate whether a received box will contain one-way or two-way parameters.

Now that you’re truly an expert on boxes, it’s time to get into a whole new dimension of storage…

# CHAPTER 15

## Flocking to Gaggles for Style, Flexibility, and Fun



Commands Introduced:

## ----- Teacher's Journal -----

'This is going to take forever!" I heard Luis complaining from across the room.

"C'mon!" urged Rina. "We need, like, a hundred of them–a THOUSAND!"

"What??" Luis exclaimed. "You're crazy."

"No way!," Rina retorted. As I drifted closer, she conceded, "Okay okay; then 20…AT LEAST!"

"But we've only done five so far and that's just constructing them and storing them! After that, we gotta load 'em all to move 'em around! You're crazy!" Luis asserted once again.

"What've you got going?" I asked, deciding it was time to intervene.

"Amoebae…KILLER AMOEBAE, taking over the world," Rina said, showing me the graphics she'd brought in.

"She wants to put about a thousand of them!" Luis said.

"I said 20, all right? But they're really SMALL, you know? It takes a lot to take over the world."

"It's going to take forever," Luis said again, shaking his head with a fatalistic expression.

"Maybe not, Luis," I said. "There's another way to store data I've been wanting to tell you guys about. It's something called 'gaggles.'"

"GAGGLES!?" Luis and Rina cut in together, making the requisite choking noises deep in their throats.

"Not like that; gaggle, you know, like flocks of geese, groups of stuff." (I glanced at Rina, who was giving Luis a 'Now-You've-Done-It!' glare) "HOARDS AND HOARDS of killer amoebae" I added, watching both faces fill with interest.

"Okay," Rina said. "What's a gaggle?"

## Bunches of Boxes

A gaggle is actually just a collection of boxes that are working as a bunch, either because they contain identical things or because they contain parts of a whole.

Each gaggle has a specific number of boxes, and each box has a number assignment according to its place in the gaggle. That means you don't have to name each individual box in the collection. Instead, you can load a particular box in a gaggle by "calling" its number. This makes for much more flexibility and better programming style. Gaggles make programs more readable, and allow us to write programs that can deal with dynamic, or shifting, amounts of data, like the number of students in your class, or the number and placement of checkers on a checkerboard.

## Two Kinds of Gaggles

When it comes to their contents, there are basically two kinds of gaggles: arrays, which are collections of things that are all of the same type, and structures, which are collections of things that are not necessarily all of the same type, but that constitute parts of a whole.

An example of an array would be a gaggle containing 6 geese flying south for the winter. An example of a structure would be a set of information containing 6 different features of any particular goose, such as its wingspan, flight speed, feather color, beak length, favorite nursery rhyme, and personal honking sound.

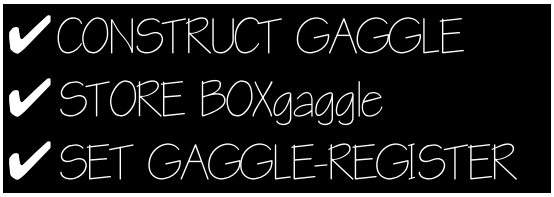
## Staking Out Your Storage Spaces

Regardless of whether it contains an array or a structure, the gaggle itself is just a big storage space. So the first thing you have to do when you want to use a gaggle is stake out some memory by constructing a gaggle of a certain size.

Constructing a new gaggle consists of four basic phases.

- First, you have to construct the number of items (or empty boxes) that you want your new gaggle to have.
- Then you name the gaggle by putting it in a box of its own.
- Next you transfer the new gaggle into the Gaggle Register, which is a separate place inside the computer that's just for gaggles. It keeps track of the whole thing so the AC is free to deal with one item at a time.
- Finally, you have to fill the open spots (or "store the items") in the gaggle. Depending upon what you fill it with, a gaggle will either be an array, with same-type things (like geese) or a structure (with different-type things like personal data features).

Since gaggles are *big* bunches of things, it takes several commands just to get one ready to go. It may seem like a lot of extra footwork at the beginning, but trust me, it'll save you many more steps later on when it comes to manipulating complex data in your program! The next few commands are the ones you need to set up a gaggle.




Remember that the gaggle itself is separate from what you ultimately put in it. When you construct a gaggle, your first task is to set up empty storage space according to your needs. The Data Wizard will show you a dialog like this:

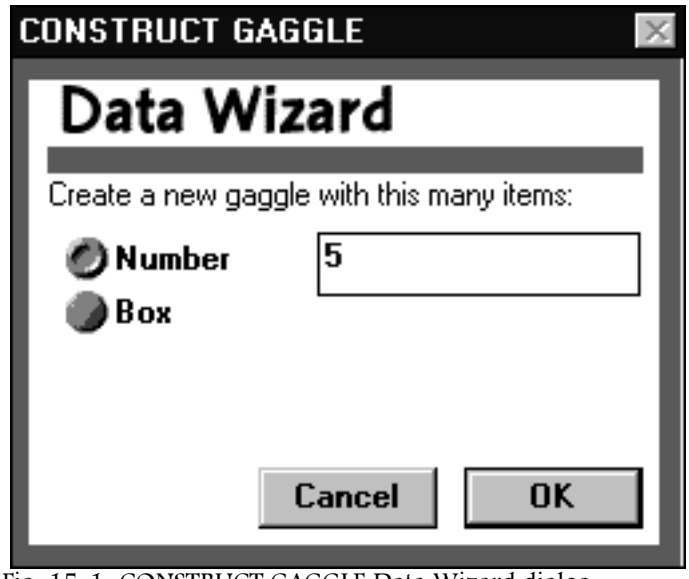



Fig. 15-1 CONSTRUCT GAGGLE Data Wizard dialog.

Let's look at the commands to complete phase one of gaggle construction in which you simply construct space for the number of items you want to store, and then bring the new gaggle into the gaggle AC, or Gaggle Register.

Look at the following sample code that completes the first step in creating a gaggle containing an array of sharks.

```
CONSTRUCT GAGGLE 5
STORE BOX sharkgaggle
SET GAGGLE REGISTER sharkgaggle
```

"Wait a minute!" you say. "What's that STORE BOX command doing there? Isn't a gaggle just a big bunch of boxes tied together? What do we need that other box for?"

## Storing Your Gaggles

Here's the thing. The command CONSTRUCT GAGGLE serves only to set aside the number of storage spaces you want for this particular new gaggle. But knowing the number of items in a gaggle is not sufficient information for identifying a gaggle later on. Let's say you want to create a gaggle of 5 fish to be eaten by your gaggle of 5 sharks, for example. Both gaggles would have the command CONSTRUCT GAGGLE 5 as the first step of their construction. You can see the problem with identification here. Trying to identify a gaggle by the number of its items would be kind of like pointing into a crowded room and saying, "My brother is the man with two legs!"

For that reason, we need to be able to identify each gaggle by something more specific. And the way to do that is to put each newly constructed gaggle in its own box, and to name the box appropriately, according to the nature of the gaggle it will contain. When you use the command SET GAGGLE -REGISTER, the Data Wizard will ask you to specify by box name the gaggle you want to fill up or use.

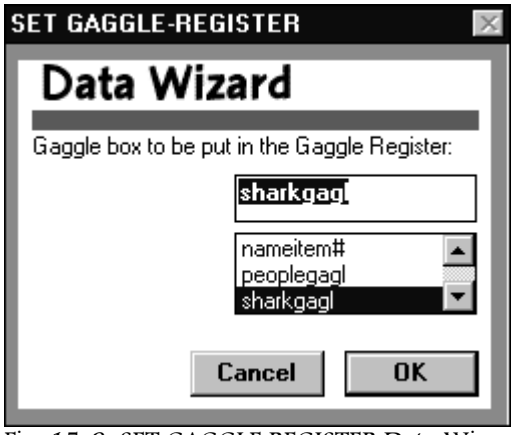
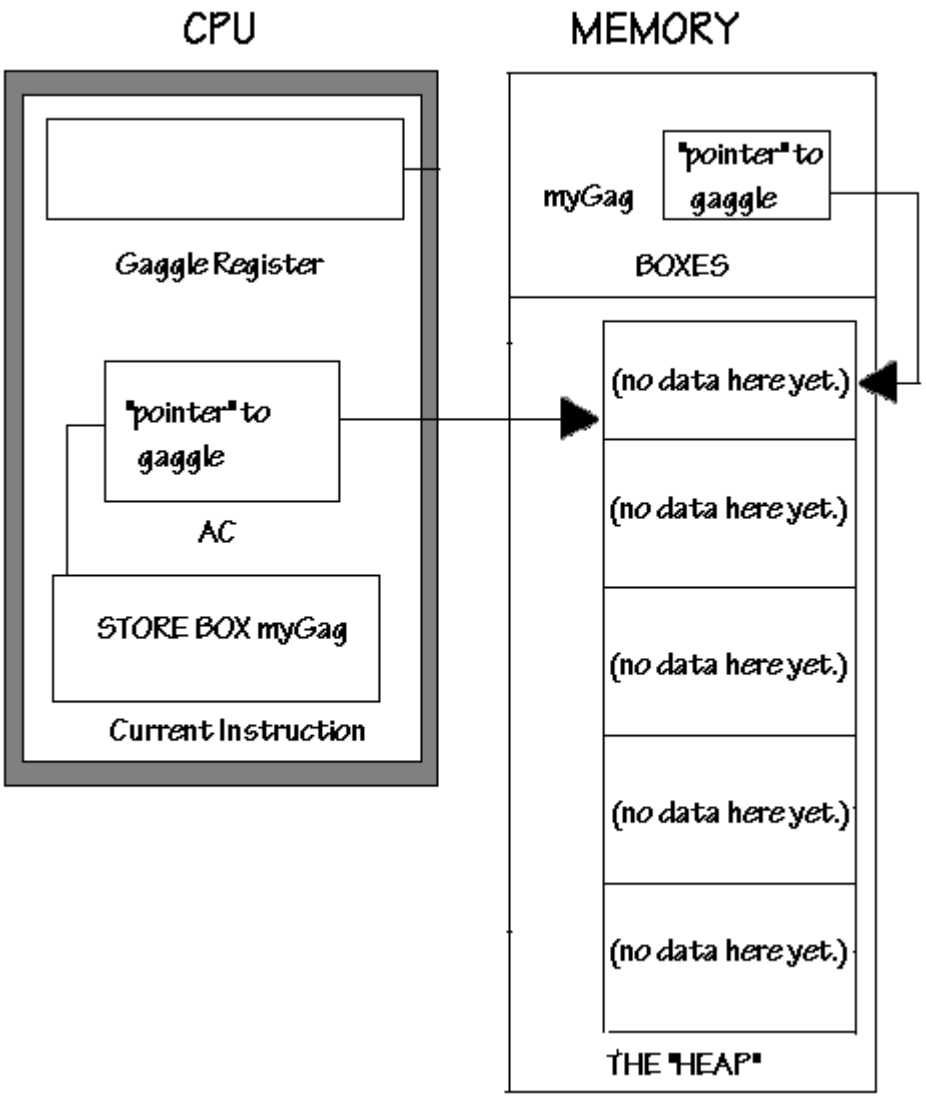



Fig. 15-2 SET GAGGLE REGISTER Data Wizard dialog.

## What's Going On Inside the Computer?

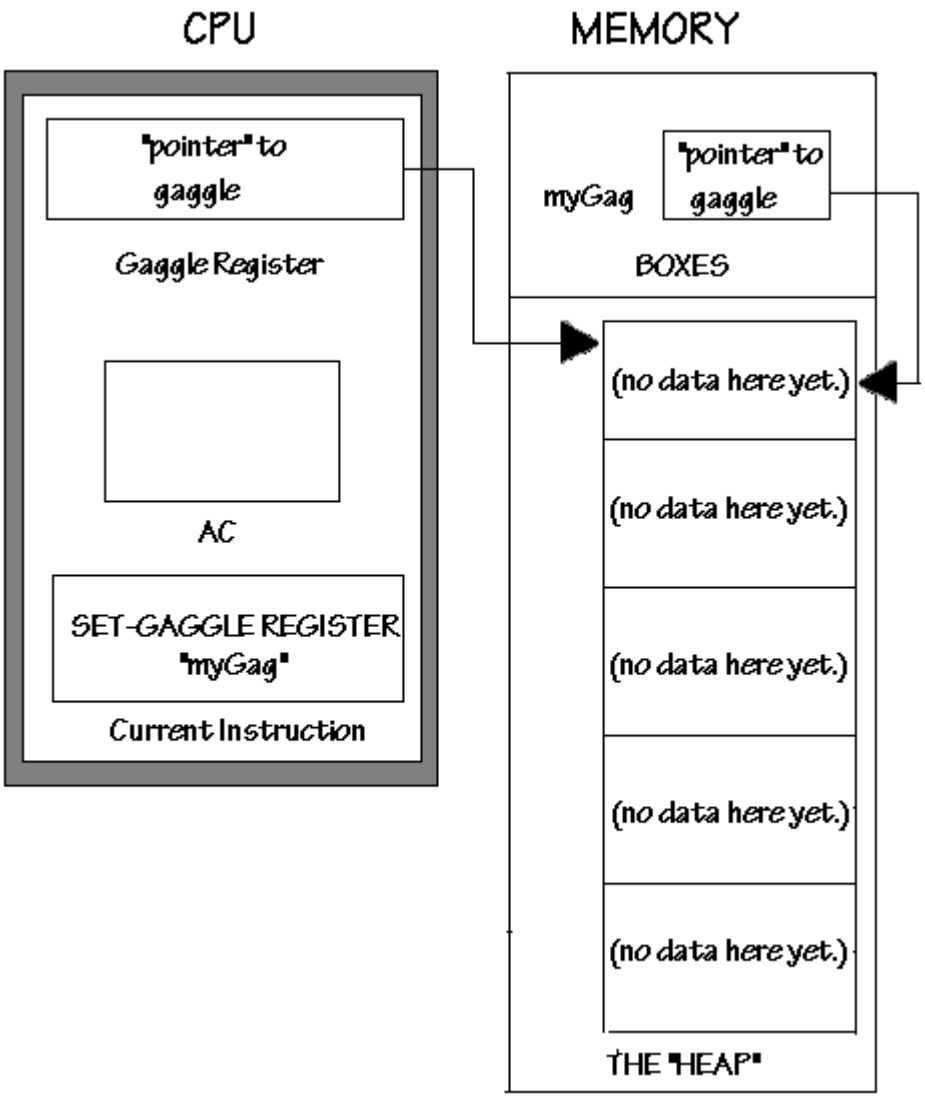
Gaggles, like objects, reside in the portion of the computer's memory called the "Heap." After you construct a given number of items for, and store, a new gaggle, a pointer to that gaggle, and not the gaggle itself, is what's in the AC.




Illus. 15A Gaggles reside in the Heap and are accessed through pointers.

But once you want to do anything with a particular item in the gaggle, you have a problem, since the AC will stop "thinking about" the gaggle as a whole, in order to deal with the individual piece of it. The pointer to the gaggle will be replaced by the data in its item, and the gaggle will be stranded, inaccessible, in the Heap.

That's where the Gaggle Register comes into play. It's a whole new part of the computer that keeps track of the entire gaggle while the AC shuffles through the gaggle's individual parts. So after you name the gaggle with the STORE BOX instruction, you have to use the command SET GAGGLE REGISTER to transfer the gaggles pointer into the Gaggle Register and free up the AC to deal with the data to be stored in the gaggle items.




Illus. 15B The Gaggle Register keeps track of the whole gaggle while the AC is busy with individual gaggle items.

## Fill 'er Up!

Everything we've done so far has only served to set aside and identify the storage space the computer will use while it runs your program. Now let's look at the next step, which is filling up, or storing the items of the new gaggle. The way this part works depends upon whether the gaggle represents an array or a structure. The first set of new commands can help you set up gaggles that hold arrays.

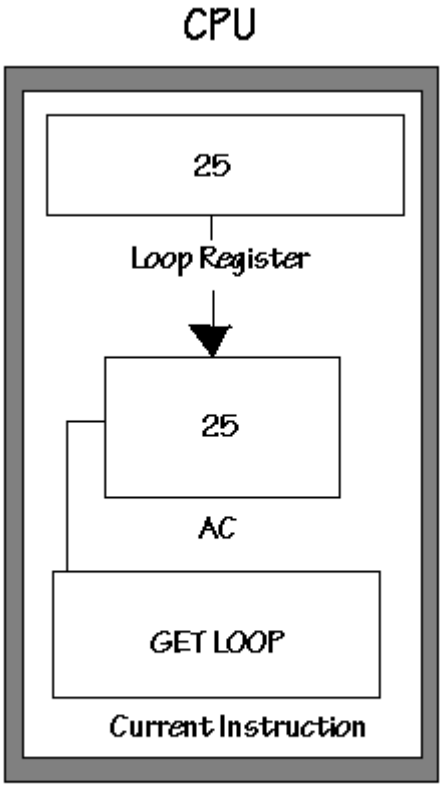
✔SET LOOP
✔GET LOOP
✔STORE-ITEM GAGGLE

Let's look at the complete code for constructing a bunch of sharks. You'll see an old command, SET LOOP, working together with its counterpart GET LOOP to do a new trick.

```
CONSTRUCT GAGGLE 5                  -constructs space for 5 items in the new gaggle
STORE BOX sharkgaggle               -stores this 5-item gaggle in a box labeled sharkgaggle.
SET GAGGLE-REGISTER sharkgaggle     -puts the gaggle in the Gaggle Register
SET LOOP 5                          -tells the computer to repeat the following action exactly one time for each item in the gaggle

@makesharks
  GET LOOP                          -puts the number on the loop Reg. in the AC
  STORE BOX sharkgagitem#           -stores number as a gaggle item number
  CONSTRUCT OBJECT shark            -constructs shark object

  PLACE OBJECT (0,0)
  SHOW OBJECT
  STORE- ITEM GAGGLE sharkgagitem#  -puts the shark in  gaggle item corresponding to the current number on the Loop Reg.
  JUMP LOOP @makesharks             -repeats until Loop Reg. is 0
```

## What's Going On Inside the Computer?

Remember that SET LOOP sets a counter, or Register, to tell the computer how many times you want something be repeated. The Loop Register always contains a number that represents the number of loops, or repetitions, left to be completed. So if you use SET LOOP 5, for example, the Loop Register contains first 5, then 4, then 3, then 2, then 1, then 0, with the numbers dropping (or, in techie speak, "decrementing") after every repetition. The command GET LOOP simply tells the computer to take the number currently in the Loop Register and put that number into the AC.




Illus. 15C GET LOOP puts the current number on the loop register into the AC.

## Avoiding Object Stacks

Actually, there's something wrong with the section of code above…Can you find the bug? Here's a hint: If you typed it in and ran it exactly as it appears, you'd only be able to see one shark. Why?

Because you told the computer to put all of the sharks at exactly the same location, that's why! You do have 5 sharks; it's just that they're all stacked up on top of each other, so you can only see the one on top.

When you make gaggles with objects, you'll always want to make a variable out of at least one of the coordinates, so that each of the sharks ends up in a different place. You can do this by storing the coordinate in a box, and then each time the computer runs through the loop, it adds to the coordinate a number that's big enough to cover the length of the object, plus a little elbow room. A comfortable distance from one shark bottom to the next, for example, is about 20 on the Playground grid. Here's how the corrected code looks:

```
LOAD NUMBER 0
STORE BOX sharkbottom

CONSTRUCT GAGGLE 5
STORE BOX sharkgaggle

SET GAGGLE-REGISTER sharkgaggle

SET LOOP 5
@makesharks
GET LOOP
STORE BOX sharkgagitem#

LOAD BOX sharkbottom
ADD NUMBER 20
STORE BOX sharkbottom

CONSTRUCT OBJECT shark
PLACE OBJECT (0,sharkbottom)
SHOW OBJECT
STORE-ITEM GAGGLE sharkgagitem#

JUMP LOOP @makesharks
```

Notice that when we GET LOOP here, we store the number in a box named "sharkgagitem#." That's because the number of loops exactly matches the number of items in our gaggle. This section of code is essentially saying, "with 5 sharks left to construct, make one and put it in gaggle item #5. ...With 4 sharks left to construct, make one and put it in gaggle item #4. ...With 3 sharks left to construct, make one and put it in gaggle item #3. ..."and so on, until there are no more sharks to construct and no more gaggle items left to fill.

This procedure for setting gaggle items with a set loop only works, of course, if the gaggle represents an array, meaning that the same type of thing is going into every item (e.g. item #1-goose; item #2 - goose; item #3-goose, etc.)

## What About Structures?

For gaggles representing structures in which item #1 could be "6 feet" (for wingspan), item #2 could be "10 mph" (for flight speed), and item #3 could be "Diddlediddle Dumpling" (for favorite nursery rhyme), we need to do something different.

Look at the following sample code constructing a gaggle of personal information items. See if you can figure out how the command STORE-ITEM GAGGLE fits in here.

```
LOAD NUMBER 1
STORE BOX name-item#
LOAD NUMBER 2
STORE BOX age-item#
LOAD NUMBER 3
STORE BOX color-item#

NEW GAGGLE 3
STORE BOX userinfogagl
SET GAGGLE -REGISTER userinfo.gagl

LOAD STRING "Please type in your name."
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE-ITEM GAGGLE name-item#
LOAD STRING "Please enter your age."
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE-ITEM GAGGLE age-item#

LOAD STRING "Please enter your favorite color."
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE-ITEM GAGGLE color-item#
```

Whew! There's a lot going on here! Let's work backwards, looking at this code sequence from the bottom section up. Since you already know all the string commands, it should be pretty easy to see how STORE-ITEM GAGGLE works here. Everything that the user types in during the READ-SCREEN phases of this sequence gets stored as an item in the gaggle. So this gaggle might ultimately contain "Lucille" as the first item, "13" as the second item, and "hot pink" as the third.

In the above code, the three commands making up the second section construct memory space for a three-item gaggle, name the gaggle by way of storing it in a box, and put the gaggle into the Gaggle Register.

Now let's look at the first group of commands. These may have been the most mysterious part of this code. I mean, really—what's the point of storing plain numbers in boxes and then using the box names instead of the numbers later on? Why not just say STORE-ITEM GAGGLE 1, 2, and 3?

## Using Box Names for Readability and Flexibility

There are a couple of reasons. The first has to do with readability. Advanced programmers do whatever they can to make their thinking clear and "readable" in their code. This is important for both the programmer, who may leave off working on a complicated program

and come back to it later, and for others, who may want to look at a program's code and try to see what's going on. In this case, a descriptive box name is a lot more telling than a simple numeral.

The second reason is flexibility. This is just another take on how boxes serve as variables in FUNdaMENTAL. Here's how it works. At some point while working on a program, you may decide to add an item, or change the order of the items in a gaggle. If you did that, and your numbers were not stored in boxes, you'd have to go through and change the numbers every time an affected gaggle item was mentioned in the code. (For example, if you wanted to make "favorite color" item #4, and stick in "best family cooking secret" as item #3, you'd have to go through your program and change every reference to the color item from a 3 to a 4).

When you use the command STORE-ITEM GAGGLE, you'll see a dialog offering you a choice between assigning a number or choosing a box that contains a reference number.

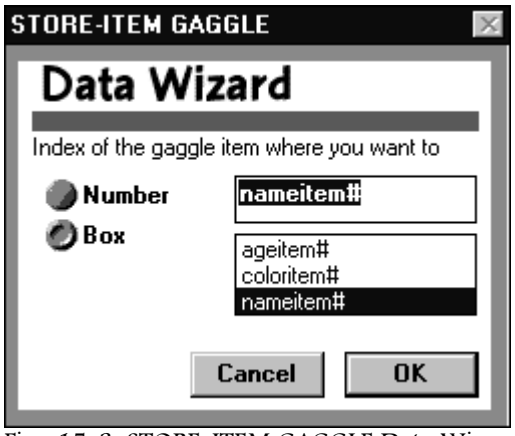



Fig. 15-3 STORE-ITEM GAGGLE Data Wizard dialog.

For gaggles that represent structures, it's always best to start, as the above code does, by storing one number for each gaggle item in a box. That way, all the boxes will be there to choose from when you use STORE-ITEM GAGGLE. (Make sure, as always, that the box has a good, descriptive name on it. And it's a good idea to add something like "item#" or "index" to the end of the box name, so you can tell it apart from any regular, single boxes. They'll all appear in the same list!)

## Now You've Got a Gaggle

So far, we've taken time for some pretty fancy footwork just to construct gaggles. Now it's time to see how the gaggles of data that you've stored away can be manipulated to achieve different program goals.

Let's go back to the example of the array gaggle containing a bunch of sharks. For the sake of adventure, let's assume that this gaggle contains 100 sharks, instead of just 5. And now that we have them, we want to get them swimming. Here's the code.

```
  SET LOOP 10
@sharkattack
  SET GAGGLE -REGISTER sharkgaggle
  SET LOOP 100
@swim!
  GET LOOP
  STORE BOX sharkgagitem# (This stores a loop reg. number in a
                                box called sharkgagitem #)
  LOAD-ITEM GAGGLE sharkgagitem#
  MOVE OBJECT (5,5)
  MOVE OBJECT (5,-5)
  JUMP LOOP @swim!
  JUMP LOOP @sharkattack
```

### ✔ LOAD-ITEM GAGGLE

Like STORE-ITEM GAGGLE, LOAD-ITEM GAGGLE calls up one of the items of a gaggle. But while STORE-ITEM GAGGLE "fills-up" an empty item space with some kind of data, LOAD-ITEM GAGGLE "takes out" the contents of the specified gaggle item and loads it into the AC, to be manipulated by whatever commands follow. The number in the Loop Register points to the item of the same number in the gaggle. As the numbers drop down, each new gaggle item is taken out and moved automatically. The loop at the start marked with "@sharkattack" ensures that each shark swims forward a total of 10 separate moves.

Consider now the difference between using LOAD-ITEM GAGGLE in a set loop of 100, and writing out

LOAD BOX shark1
MOVE OBJECT(5,5)
MOVE OBJECT (5,-5)
LOAD BOX shark2
MOVE OBJECT (5,5)
MOVE OBJECT (5,-5)
LOAD BOX shark3...

until all 100 sharks have been taken out and moved!

All of a sudden the effort of constructing storage space for all those sharks in a gaggle doesn't seem so bad after all! (And that's not even to mention the yards of code it would take to construct and store each shark individually in its own box.)

## LET'S PROGRAM!

Okay. Take a deep breath; it's time to try some of this stuff. You are going to make a new animated program with froggies, only this time you'll use a gaggle array to make an amphibious plague of biblical proportions!

1. Launch FM, and open Program8.fmp. At the FM Welcome window, select Open Existing Programs, and navigate through c:\, Fundam, Manual, Practice, and Program 8, and open the Program8.fmp.

   This program has a frog object already designed. You need to use gaggle commands to make 10 frogs and get them hopping in a wave across the Playground.

2. Use the instruction LOAD NUMBER 0, followed by STORE BOX frogbottom so that all your frogs don't end up stacked, 15 thick.

3. Use CONSTRUCT GAGGLE 10, followed by STORE BOX. Don't forget to give the box an appropriately amphibious name.

4. Use SET GAGGLE~ REGISTER and specify the name of the box in which you just placed the new gaggle.

5. Use SET LOOP so that the computer will keep making frogs until all the gaggle items are filled. (The loop should have the same number of turns as there are items in the gaggle.)

6. Place a marker at the spot to which the computer should return in order to "makefrogs."

7. Use GET LOOP, followed by STORE BOX. A good name for this box would be "frogagitem#" or something like that.

8. Use LOAD BOX frogbottom, followed by ADD NUMBER 20, and STORE BOX frogbottom, so that each frog's bottom will be placed a distance of 20 from the previously constructed frog.

9. Construct the frog object and place it at (0, frogbottom). Then show the object.

10. Use STORE-ITEM GAGGLE and specify the box "frogagitem#," which contains the number corresponding to the current gaggle item.

11. Use JUMP LOOP, so the computer will do it all again!

12. Now you've got a gaggle of froggies! Get them hopping! Use the sample shark code from earlier in this chapter to see how to use LOAD-ITEM GAGGLE to make each one hop. Set another loop so they keep hopping. And don't forget to add some MORPH commands so the frog will stretch out and bunch up between hops.

## Programming Tips and Tricks

Just because you stored the items of your gaggle in sequential order doesn't mean you have to load them back into the AC that way. If you're ready for a challenge, try replacing GET LOOP with RANDOM NUMBER to make an exciting little frog race that has a different outcome each time.

When you're using gaggles, the Debugger really earns it's keep. On your previous visits to the Debugger window, you may have noticed a little field labeled "gaggle." This shows you the contents of the Gaggle-Register. Until the first SET GAGGLE REGISTER instruction is executed, this field will display the word "unknown." But once the Gaggle Register contains a gaggle, this field will reflect that with a reading that looks something like this: "a gaggle of 5 elements." Once a gaggle shows in this field, you can make use of the debugger's Gaggle window. Open it by double-clicking on the gaggle field in the Debugger window. It looks like this:

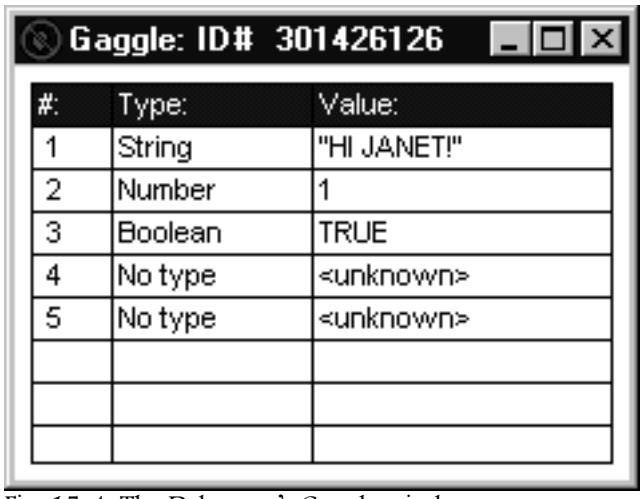



Fig. 15-4 The Debugger's Gaggle window

As you step through your program, you'll see the contents of each item displayed with its data type in the grid. Expert programmers rely on this to help them keep track of what's what when they get up to their necks in gangly gaggles.

## How Many Geese in a Gaggle?

Once you understand the benefits of gaggles, and know how to use them, it's likely that you'll have more than one of them in a program. As we have seen, if you want to get anything done with a gaggle, it's essential to know how many items are in it. Otherwise, you won't know how many loops to set and get in order to make sure that all the items (or boxes) in a gaggle are accounted for.

But with more than one gaggle going, it may be difficult to remember at any given time how many items a particular gaggle has. This is especially true if you've been making changes as you've been going along, adding or subtracting items in your gaggles. Remember that when you tell the computer to SET GAGGLE-REGISTER, it will only show you the existing gaggles by name without reminding you how many items each gaggle has.

### ✔COUNT GAGGLE

With COUNT GAGGLE you can make sure that the computer knows how many items it's dealing with. When you use this command, the computer automatically counts the number of items in the gaggle currently in the Gaggle Register, and puts the resulting number in the AC. The following sample code demonstrates how it works.

```
  SET GAGGLE-REGISTER sharkgaggle
  COUNT GAGGLE
  STORE BOX numsharks
  SET LOOP numsharks
@swim
  GET LOOP
  STORE BOX sharkgagitem#
  LOAD-ITEM GAGGLE sharkgagitem#
  MOVE OBJECT…etc.
```

In this case, we don't even have to know the exact number of items, because the computer does. All we need is a box called "number of sharks" (or "numsharks") that is read to take up whatever number is put in the AC by the COUNT GAGGLE command. Then, we're in business.

### ✔DESTROY GAGGLE

Gaggles, like objects, take up computer memory in the Heap while your program is running. That's fine, as long as you're getting good use out of them. But if, as your program progresses, a once-important gaggle fades from the scene, you want to make sure that it's not still skulking out of view, wasting memory. It's a good idea to get in the habit of using the command DESTROY GAGGLE in your programs with gaggles. Use this command once for each gaggle somewhere before the command EXIT PROGRAM. Even if you've only got one gaggle going, you'll still be practicing good, efficient programming if you destroy it at the end.

## Arrays of Arrays

The real power of using gaggles in your programs becomes apparent when you are ready to use gaggles of gaggles! (No, I'm not kidding.)

It is possible to have a gaggle in which one or more of its items are, themselves, gaggles. There are two main reasons to do this.

The first is to make an array of arrays. This makes the most sense when you think about it in terms of representing a grid or chart with columns and rows. If you think of each column or row as a gaggle in itself, (a set of identical cells or squares), then the whole chart is a gaggle of these gaggles.

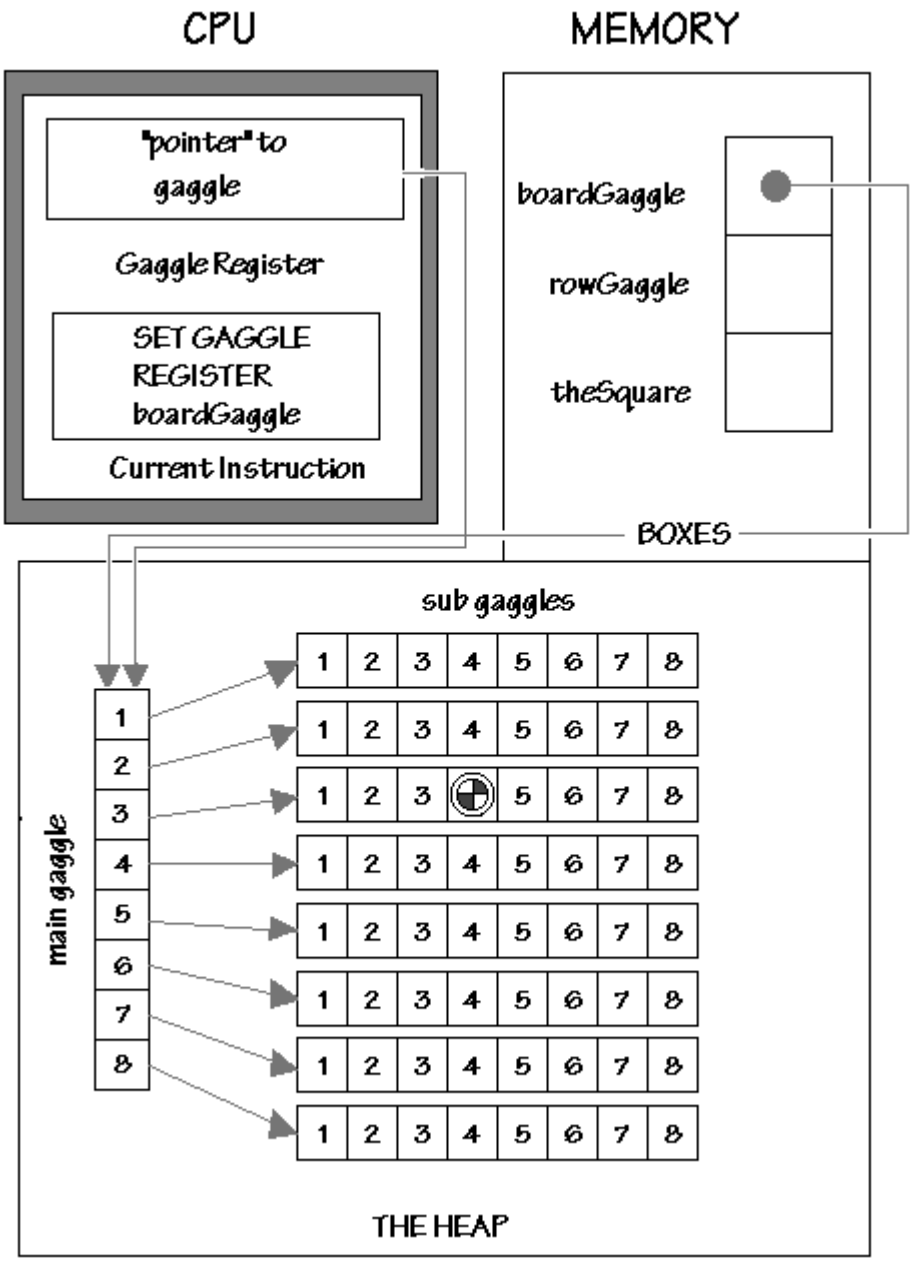
This is useful if you want to make a game of checkers, and you need to keep track of which piece is on which square. Each square could be a number that corresponds to red checker, black checker, or no-checker. Each row of the board would be represented by a gaggle of 8 squares. Then you would want to have a larger gaggle of 8 row gaggles for the whole board.

To find out what's on the fourth square in the third row, your code would look like this:

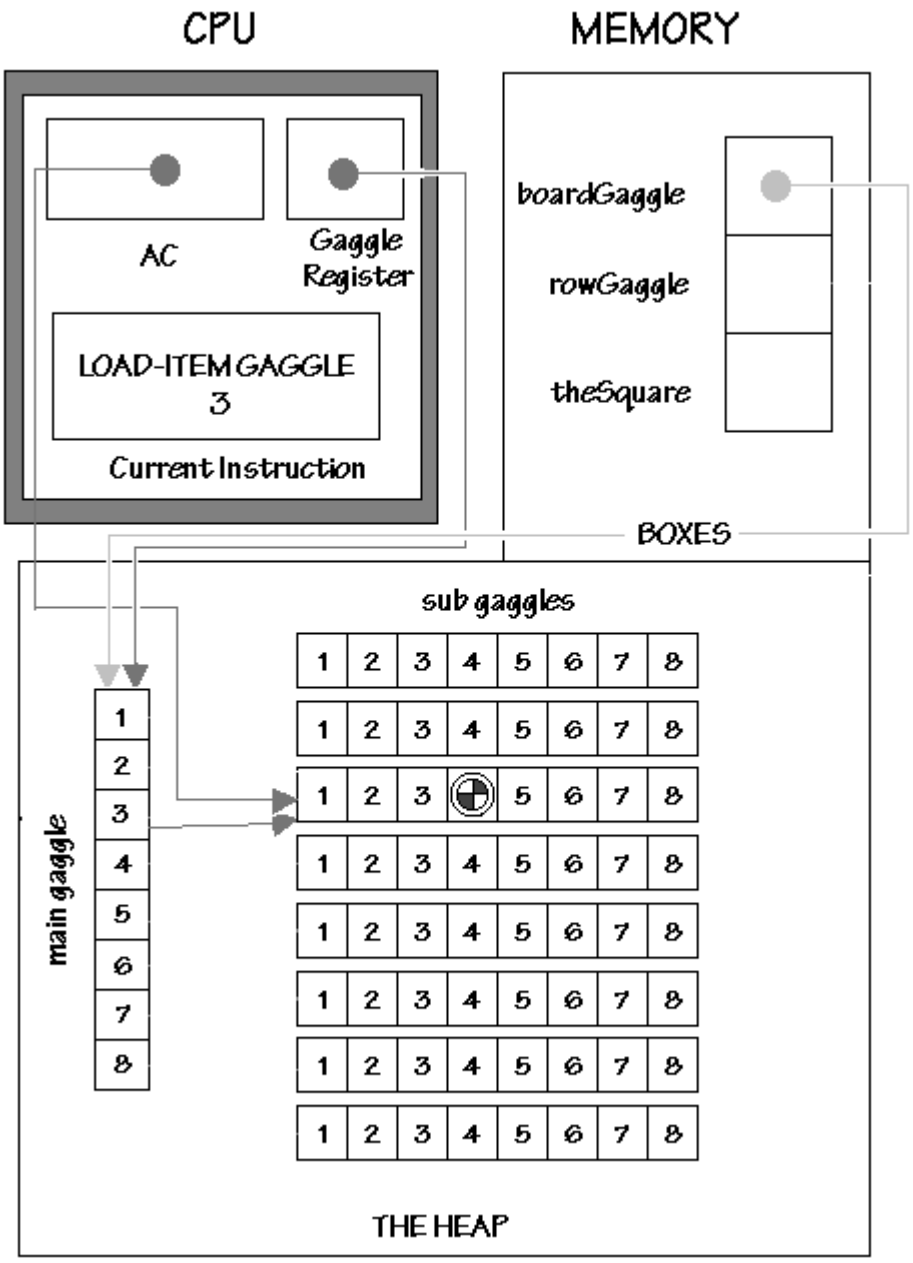
```
SET GAGGLE -REGISTER boardgaggle
LOAD-ITEM GAGGLE 3
STORE BOX rowgaggle
SET GAGGLE -REGISTER rowgaggle
LOAD-ITEM GAGGLE 4
STORE BOX the square
```

## What's Going On Inside the Computer?

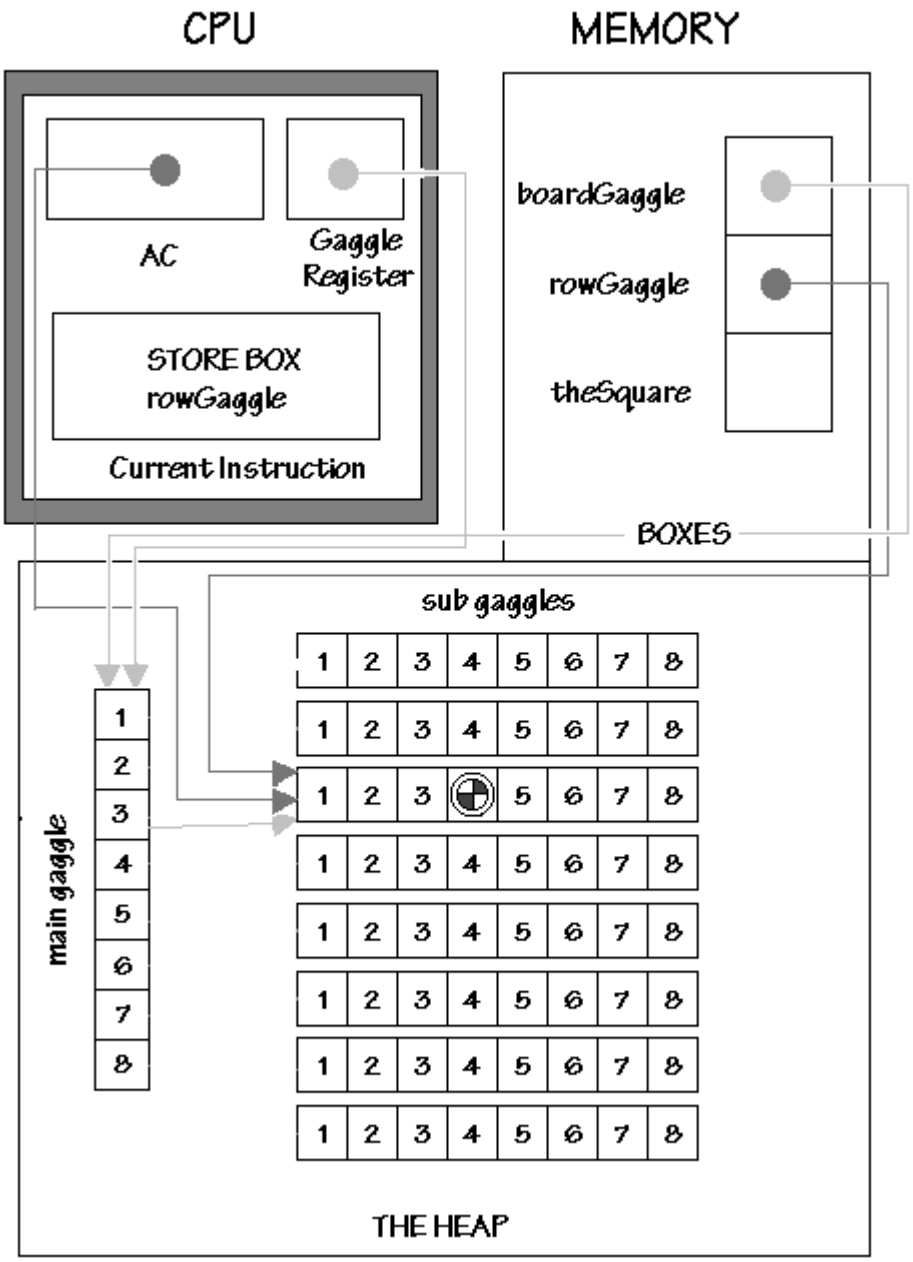
Once you get into 2-D gaggles, it's easy to lose track of what does what. Look at the following illustrations for a pictorial blow-by-blow (or, rather, command-by-command) of the code sample above.



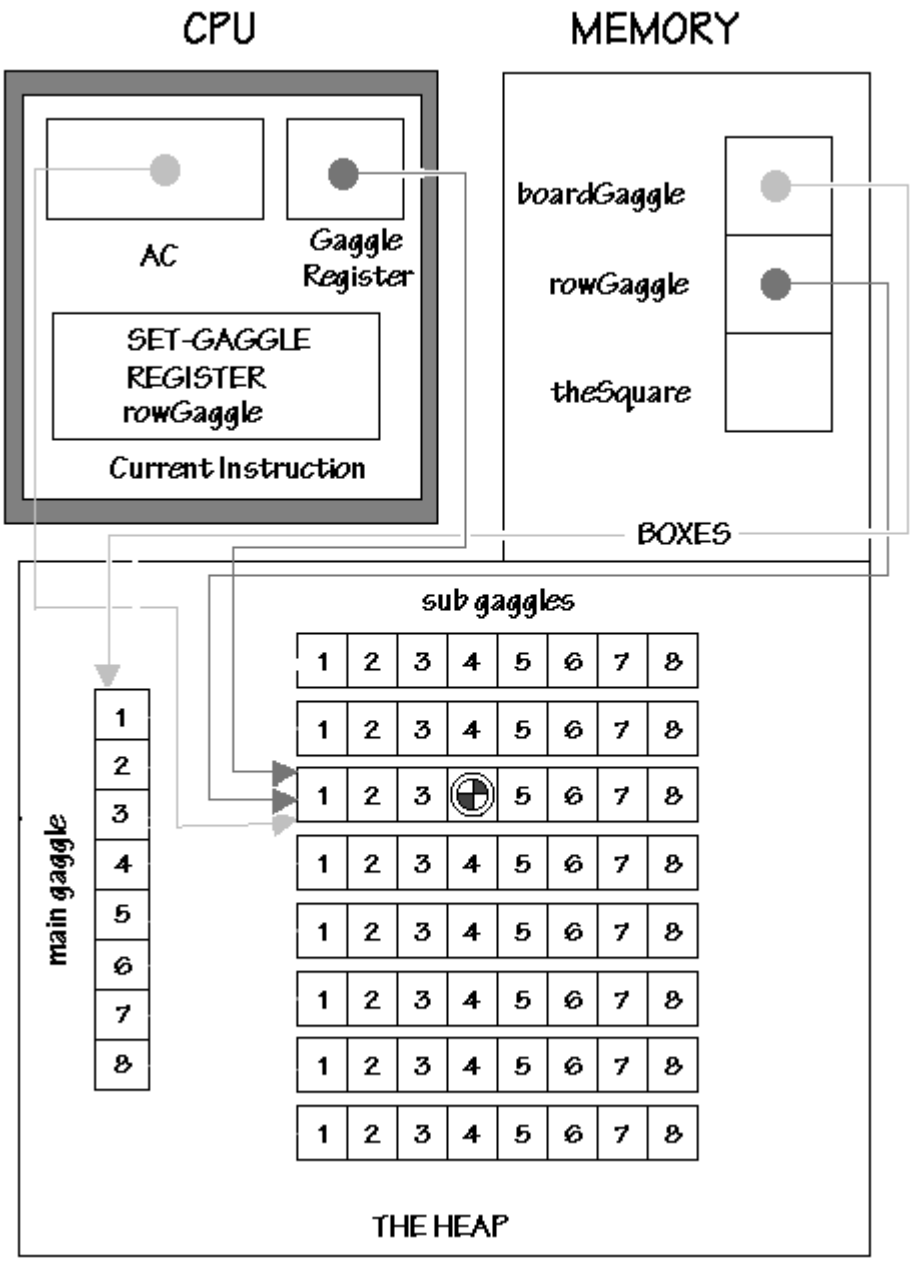

Illus. 15D SET GAGGLE REGISTER boardgaggle.



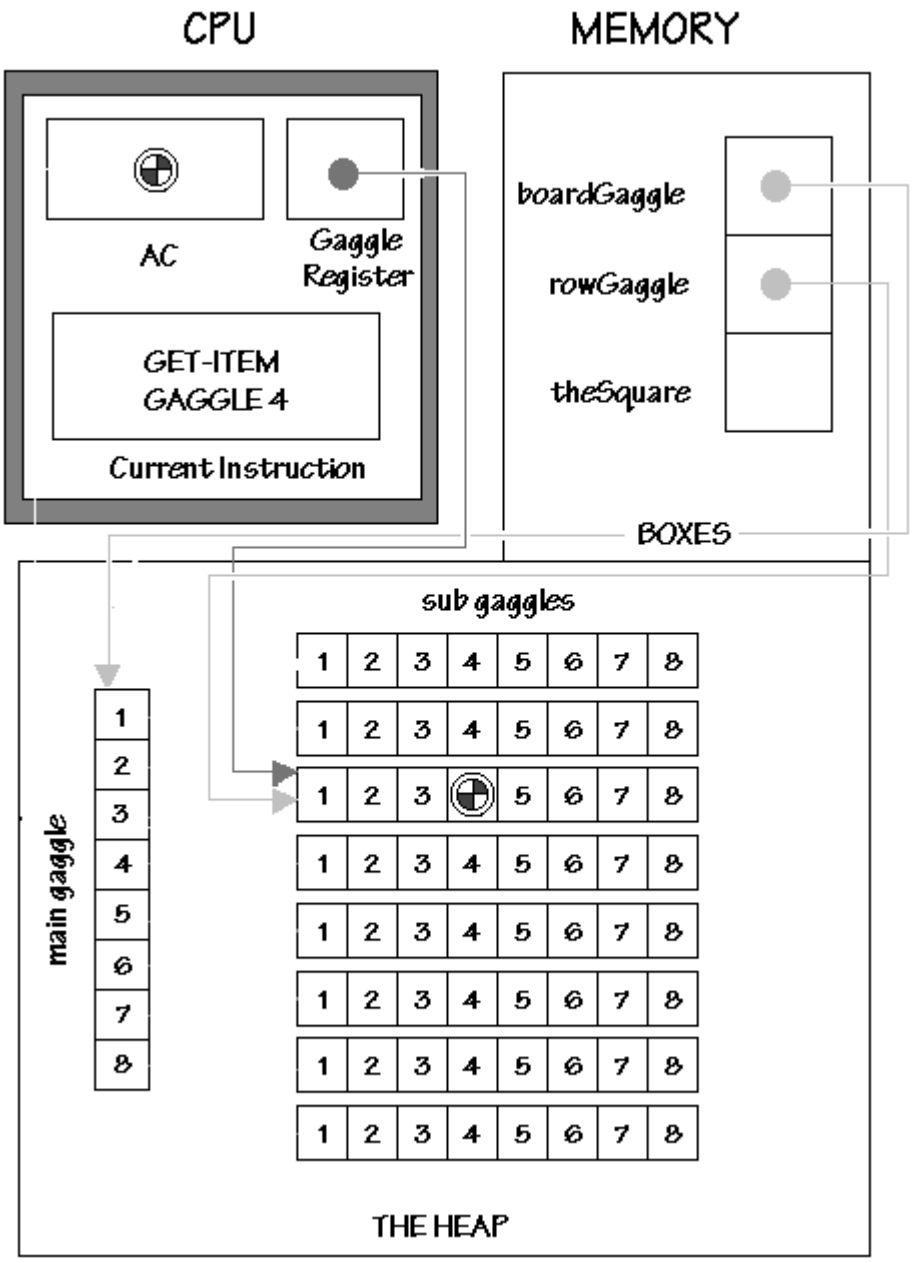

Illus. 15E LOAD-ITEM GAGGLE 3




Illus. 15F STORE BOX rowgaggle




Illus. 15G SET GAGGLE REGISTER rowgaggle




Illus. 15H LOAD-ITEM GAGGLE 4

## Arrays of Structures

The second reason to have a gaggle of gaggles is to make an array of structures. Remember that a structure is a linked set of dissimilar items, such as the personal data items for one individual.

Say you want to store information about your whole flock of geese. You have 23 geese in the flock. For each goose you want to keep track of its wingspan (a number), feather color (a string), and personal honk (a sound). So you make a structure for your goose: a gaggle with the wingspan, the feather color, and the personal honk as its items. All set, right? Wrong! Because now you want to link this gaggle to 22 other gaggles, each containing the same kind of

information about another goose in the flock. In this case, the larger gaggle would form an array because it would be a collection of structures that are all of the same type.

Looking back at the sample checkers code above, can you see how you'd find out the wingspan of the 18th goose in your gaggle?

And guess what! You can even make 3-dimensional gaggles as well (that is, gaggles of gaggles of gaggles of gaggles!)...but that's a whole other book!

Whatever dimension of gaggles you're exploring, remember: When you've got more than one gaggle going, MAKE SURE TO SET YOUR GAGGLE REGISTER TO THE NEXT GAGGLE. Otherwise, you'll never get your goose!

# CHAPTER 16
## HOLD THAT THOUGHT!
## Using FILE Commands to Save Information



Commands Introduced:

## ----- Teacher's Journal -----

Josh and Thai had spent the period making a fairly involved MadLibs program. Based upon a section of the United States Constitution taken from their American history textbook, the program substituted crucial nouns and verbs with those chosen blindly by their program's users, with hilarious results. The initial demos had been a huge hit with their classroom compatriots.

But now they were calling me over to ask for help.

"It won't print!" Josh said, trying the Control Key-P combination with exasperated pokes.

"What, the task?" I asked, still not up to speed on the problem.

"No, this!" Josh said, indicating the latest Crazy Constitution that had been left in the Conversation window by a fellow student.

"It's hilarious. We want to keep it. We want to keep them all," said Thai.

"Oh, I get it," I said.

"But it won't print, and as soon as the program's run again it's, you know, just gone," said Josh.

"I guess we'll just have to copy them all out on paper, all the good ones, anyways," said Thai, reaching resolutely for his backpack to take out a pencil and paper.

"Hold on, Thai. I think there's a better way to do this," I said. "FUNdaMENTAL knows how to save strings and numbers in a text file on the hard disk. You can open the files up through the notepad application on your computer and print them out from there."

"We can?"

"Yup," I said. "But you'll have to add some FILE commands to your program."

## Long-Term Parking

When you write an instruction like LOAD STRING "what's your name?" the command, along with its string-data input is stored away on the computer's hard drive as soon as you save the program. But if your user uses the commands WRITE-SCREEN STRING/READ-SCREEN STRING, and then types in "Irving," that name string data will then exist in the computer's memory only for as long as the program is running. When the program ends, it's gone, and it definitely won't be there after you turn off the computer and come back again tomorrow.

This is where files come in. With a file, you can save text information between executions of your program, between the times you use FUNdaMENTAL, and even after you've turned off your computer!


You can file away strings and numbers using commands almost identical to the ones we use to move strings in and out of the Conversation window: WRITE-FILE STRING and WRITE-FILE NUMBER. The code might look something like this:

```
LOAD STRING "what's your name"
WRITE-SCREEN STRING
READ-SCREEN STRING
WRITE-FILE STRING "namefile.txt"
```

With this code, the user's name has now been stored on the hard drive, in a text file called "namefile.txt". It won't go away, even when the program ends and the computer is turned off.

Notice that when you use the WRITE-FILE commands, the input data is not the thing to be put in the file, but rather the *file name.* But here's the tricky part. Whenever you give the computer an instruction with a data input, the data has to be of a type that the computer recognizes. Trouble is, there is no such data type as "file name." The only kinds of data FM recognizes are strings, numbers, objects, sounds, booleans and gaggles. (And I promise you that neither of the last two data types actually means "file name" in Martianese!)

How can we get around this? Well, what is a name, anyway? It's a string of characters, of course. So the way FUNdaMENTAL recognizes and works with file names is to have you convert them into strings in their own right.

When you use the command WRITE-FILE STRING, a Data Wizard dialog that looks like this will appear:


Fig. 16-1 WRITE-FILE STRING Data Wizard dialog.

Notice that you have a choice here between specifying a file name in the form of a string, or a specifying a box. Don't get confused! The string that you want to put in the file is already identified and waiting in the AC. The string the Data Wizard is asking for here is the one that will serve as the name of the file you're setting up. All these file-name strings will need to have a name or eight characters of less, followed by ".txt". They should look something like this: "namefile.txt", or "madlib.txt".

Note that you can also use the LOAD STRING command with these file names. You can have a file name with a ".txt" ending as the input, and then store that file-name string in a box. Then, when you use the WRITE-SCREEN command, you can click on the box radio button in the Data Wizard, and then select the box that holds your file-name string. Either way, the contents of the AC get stored away on the computer's hard drive.

## Line 'Em Up!

Once a file is set up, it can hold as many strings or numbers as you want. For example, if Irving, Mathilda, Serafina and Hank all walked by and used a program like the one in the above code sample, then somewhere on the hard drive you would have a text file called "namefile.txt" containing the following:

Irving
Mathilda
Serafina
Hank

Each time the WRITE-FILE command is executed for a particular file, the data in the AC is stored on the next line down in the text for that file.

## Dipping into Files

Getting things into permanent files is useful for a variety of reasons. For one thing, it's a way to get a hard copy printout of program output as it appears in the Conversation window. But there may be other times when it is useful for the computer to pull things out of an already existing file in order to use them during the program. And, of course, you can.

✔READ-FILE STRING
✔READ-FILE NUMBER

READ-FILE STRING and READ-FILE NUMBER are two commands which allow you to pull string and number data out of existing ".txt" files on the hard-drive in order to manipulate them in your program. When you use a READ-FILE command to access either a string or a number, you'll see a Data Wizard dialog that looks like this:


Fig. 16-2 READ-FILE STRING Data Wizard dialog.

Notice, once again, that the thing you need to specify here is the *file name* and *not* the string you're trying to get out of the file. You can either type in the actual file name, in the form of a string ("namefile.txt"), or you can specify the box that holds the file- name information. Either way, the computer will open the specified file and "read in" to the AC a string or a number from that file.

But wait!...

## How Does It Know What to "Read In"?

Good question. Let's explore it further. With the WRITE-FILE commands, there's always a particular string or number waiting in the AC to be written out to the specified file. But with the READ-FILE commands, the string or number we need is already stored away in the file, and if it's in there with a bunch of other strings and numbers, then how will we know which of those strings or numbers we're getting with the READ-FILE command?

Here's how it works: Unless you tell it otherwise, the computer will begin by reading the last thing it wrote out to a file. Subsequent executions of the READ-FILE command for that file would then go up to the top line of the file and move down in order.

For example, in the "namefile.txt" we looked at in the above sample code, the order of reading priority would go like this:

Hank
Irving
Mathilda
Serafina
Hank
Irving
Mathilda
Serafina, etc.

The first READ command got the last entry in the file, because that's the last place the computer left off in working with that file. After that, it has nothing to read from below, so it hops to the top of the list.

### ✔REWIND FILE

Can you guess what this does? You got it! It gives you a way to tell the computer to go back to the top of a particular file, which allows you to make sure that your first READ command pulls information from the top of the list. In the scenario above, aREWIND- FILE command would allow you to be sure that Irving held onto his rightful place at the top of the list when the information in the file was read back into the AC. In other words, it allows you some measure of control over what the computer pulls out of the file that you specify with the READ-FILE commands.

When you use the command REWIND FILE, a Data Wizard dialog that looks like this will appear:

Fig. 16-3 REWIND FILE Data Wizard dialog.

Notice once again that you specify not the actual information you want, but rather the name of the file that holds the information.

## Where Are We Going with All This?

Like many programming concepts and skills, files are best learned with simple examples, but they don't really show their stuff until they get to work in some pretty complex programs.

In the second part of this chapter, we'll take a look at a way to write a complex MadLibs program in which the user substitutes up to 15 words in a famous literary passage. Although there are about 50 separate strings involved (including those that ask the user for nouns; verbs, etc. those that are the user's responses; and those that constitute the intact text to which the user's responses are appended) *the program only uses the command LOAD STRING twice.* (Okay, three times, but the third string is "\\", so it doesn't really count!)

Before we see how FILE commands make this possible, let's spend some time playing around with them in a simple exercise, so you can get a basic, hands-on feel for how they work before we get into the art of filing.

You're about to fill in the missing file instructions in a program that will test users on their knowledge of world capitals. The program will offer the user two options: to set up a new quiz by entering new countries and capitals into a text file, or to take the quiz. There are actually several sub-tasks involved here, but you'll only need to complete three of them to get this program up and running. Everything else is ready to go.

# LET'S PROGRAM!

You'll be working in three different sub-tasks to complete this exercise. You'll start by completing the code that allows the user to enter new countries and capitals into the file to make a fresh quiz. Next, you'll work on the code that allows a user to take the quiz on the information currently in the file. Finally, you'll complete the sub-task that is responsible for checking the user'squiz answers to make sure that the countries are all matched with the correct capitals.

1. Launch FUNdaMENTAL and open Program9.fmp. From the Welcome window, select Open Existing Programs, and navigate through the following directories: c:\Fundam\Manual\Practice\Program9.

2. Open the sub-task called "writfile." Use the Window menu on the Toolbar and select Program window. When the Program window comes up, double-click on the task name "writfile" in the task list, and the Task window for this sub-task will open up. You'll see four lines of code and a label responsible for deciding whether to execute or skip the code you're about to write, depending upon whether or not the user chooses to enter new countries and capitals. (This will make more sense after you complete the program and see how it runs. I promise!)

3. Stop and think about the code you'll need to add in order to ask the user to enter 5 countries and their capitals, and write this input out to a text file.

4. Use the command, SET LOOP 5 under the "@continue" marker. Here, you're setting up the asking and filing action to happen 5 times, once for each country/capital pair you elicit.

5. Place a marker under the SET LOOP command to mark the beginning of the asking/filing code. You might use the marker name "@ask/file."

6. Use LOAD STRING and create a string that asks the user to enter the name of a country. ("Please enter the name of a country," or some such thing, will do nicely.)

7. Use WRITE-SCREEN STRING and READ-SCREEN STRING to get your message out on the screen and the user input into the AC.

8. Use APPEND STRING "\". This tells the computer to put the name of the country on its own line when it writes it out to the text file.

9. Use WRITE-FILE STRING "capital.txt". Remember, the string input here specifies the *name of the text file* to which you want to write the country name the user put into the AC. The first time the computer encounters this instruction in executing your

program, it will automatically set up a new text file of this name in the program folder. (Nifty, huh?)

10. Drop down a line in your Instruction list to make a blank line between the WRITE-FILE instruction and the next instruction that you're about to write. (You're about to move onto asking for the capital of the country, so it makes your program more readable if you set this off from the request for a country you just completed.)

11. Repeat the same 5 lines of code, this time eliciting the country's capital. The only thing you'll need to change is the string text, which here should read something like this: "Please enter its capital." Everything else should be the same, and the capital name will be written out to the same text file, on the line just below its country.

12. Leave one more blank line, and then use JUMP LOOP @ask/file. This will cause the computer to repeat the code until 5 country/capital pairs have been written out to your "capital.txt" text file.

13. Use END SUB-TASK.

The text file that results from the execution of this sub-task might look something like this:

Egypt
Cairo
France
Paris
England
London
Taiwan
Taipei
Peru
Lima

Later in the chapter you'll learn how to go and see your text files for yourself. For now, just trust me– it's there! Now it's time to use READ-FILE commands to complete the sub-task that reads the country names out of the text file, "capital.txt" in order to test the user's knowledge of capitals.

1. Close the "writfile" sub-task, and open the sub-task called "dotest". You can do this by simply double-clicking on the little close box in the uppermost left corner of the Task window for this sub-task.

2. Find the place where you'll insert the code that tells the computer to read the country names out of the file. You should see four commands and a marker at the top of this task, followed

by the comment, "(The code for reading the file goes here)." You can highlight the line just below the comment to make a space for the instruction you're about to insert.

3. Use READ-FILE STRING "capital.txt". This will start reading from the top of the file, since the sub-task began with the REWIND FILE instruction. The first line of the file, and every other line after that, will always be a country name.

4. Use STORE BOX and define a box named "currentcntry". This will store the country name that you've just had the computer pull from the text file and put into the AC.

5. Use LOAD STRING "Please enter the capital of".

6. Use APPEND STRING with "currentcntry" as the input. When the Data Wizard comes up after you click the Use button for this command, click the radio button next to the word "Box," and then select the box name from the list. This will put the current country name on the end of the general request string in the AC.

7. Use WRITE-SCREEN STRING and READ-SCREEN STRING to get the request written in the Conversation window and the user's response into the AC.

8. Notice the next instruction in the task, EXECUTE SUB-TASK chkcap. This sub-task is in charge of reading all the capitals out of the file and comparing them with the user's response in order to determine if the user is correct. Let's go and fill in the file code for that sub-task, and then we'll be ready to run the program.

9. Close this sub-task, and use the Program window to open the sub-task called "chkcap".

10. Highlight the first line in the Instruction list for this sub-task. There's already some code in place down below, but we'll be filling in the first 7 instructions to complete this sub-task.

11. Use STORE BOX and define a box called "useranswer". Remember what was in the AC from the last sub-task when the EXECUTE SUB-TASK command started this one? We had just used READ-SCREEN STRING to get the user's guess at a country's capital into the AC. That response is what we're storing here.

12. Use READ-FILE STRING "capital.txt". Since this is the second READ-FILE command in the program, it will begin by reading the second line from the text file, and continue reading every even-numbered line until the program ends.

13. Use STORE BOX and define a box for this correct capital information. The box should have a name like, say, "capital."

14. Use COMPARE STRING and select the box called "useranswer", which is offered in the Data Wizard once you click the Use button.

15. Use JUMP<>@wrong. If the two strings don't match, this will cause the computer to jump to a marker that gives an appropriate message to the user. The marker and its code are already in place below.

16. Use LOAD STRING and create a good congratulatory string. If the two strings do match, then the computer will skip the above instruction, and congratulations for a correct response will be in order. Type in your string after selecting the command and clicking on the Use button.

17. Use WRITE-SCREEN STRING so the user can enjoy your congratulations.

And congratulations are in order for you too! You have now completed all the code necessary to run the program. Give it a test run and play around entering countries and capitals and then taking the quiz yourself. Experiment with right and wrong answers to see how the different possibilities come into play.

## Programming Tips and Tricks

That was quite a little trip in and out of sub-tasks, wasn't it? This program is a great one for demonstrating the virtues of sub-tasks, and good decomposition in general. After you've had a chance to play around with the program and get a feel for what it does from the user's perspective, go back and take a close look at all the sub-tasks to see how the program breaks down.

## Files in Advanced Programming

Files get really interesting to work with only in pretty complex programs. Once you get into more advanced stuff you can use file commands to your advantage in a variety of ways.

In some cases, we use files so we can keep a permanent record of user input. For example, a class database is one kind of program that makes good use of file commands. Stop and think about it. You'd be working with a two-dimensional gaggle (a big gaggle of People entries made up of smaller gaggles containing name/phone#/homework info. for each person), and you'd need a sub-task in charge of putting information into the database, and another sub-task in charge of getting information out. Of course, you'd want all the people data to be stored permanently on the hard drive, so there would be a lot of file commands involved here, wouldn't there?

In other cases, however, you can fill up a file in advance and structure it so that it supports the writing of an extremely succinct, readable, and elegant program.

## LET'S CHECK IT OUT!

The program we're about to look at is the MadLibs program I described earlier in the chapter. The first thing you should do is check it out and see what it does. (It's pretty fun to play with, so don't get carried away rewriting Shakespeare and forget to come back!)

1. Launch FM and open Demo3.fmp. You'll find it by navigating through Fundam, Manual, Examples and Demo 3 folders.

2. Play with the program and see what it does.

3. Look at the code. Start with the main task, and then look at the sub-tasks in this order: "set-up", "askinput", "writelib". If you can't see any part of the Program window to click on and bring it forward, then use the Window menu on the Toolbar to get to it. From there, you can get into the code of each of the tasks in the program by double-clicking on their names in the list. Notice that the programmer has left some comments in the code to help explain what's going on. But it's still a little hard to figure out what's what until you see the file from which all these commands are reading. So let's go take a look. Then we'll come back and study the code one more time.

4. Get into your computer's File Manager. (That's the one that lists all the directories for everything you've got going inside your computer). Notice we're leaving FM territory here so that you

can view the contents of files through your computer's notepad or word processing application.

5. Navigate through the following directories: c:\Fundam\Manual\Examples\Demo3. You do this the same way you navigate through folders to open program files in the FM Open Program directories window. (We're just entering a familiar place through a new door!)

6. Open the text file "Hamlet.txt" which is in the Demo3 directory. Once the Demo3 directory is open, you should see a files list that looks like this:

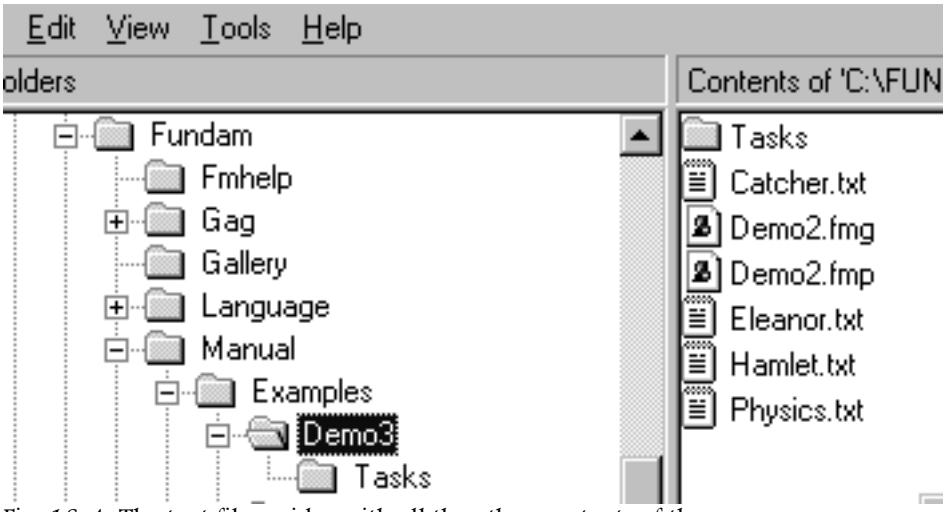


Fig. 16-4 The text file resides with all the other contents of the program folder.

7. Double-click on the text file icon marked Hamlet.txt.

8. Examine this text; print it out, if possible.

9. Return to FUNdaMENTAL and get into Demo3.fmp once more, so we can look at the file and the code together.

## The Art of Filing

You can use WRITE-FILE commands to get information into a file, but you can also manually set up a text file in advance through your computer's notepad or word processing application. (Instructions for doing that follow later in the chapter.) That's how this file was set up. In order for it to work properly, the programmer had first to envision a file *formula* that would work with the desired program design. If you go back and look at the other two files in this program, you'll see that they also follow the same formula, starting with a number, followed by a bunch of solicitations for nouns and verbs and such, and ending with a long list of literature fragments strung together between numbers. Are you starting to get a sense of where all the string data and gaggle item numbers are coming from for the

program? Let's go back and take a look at the program code one more time.

## The Art of Programming with Files

The neat thing about this program is that it works just like a little machine that can read any file set up according to the proper formula, and turn that information into a fun MadLibs game. Notice that the program contains very few specifics. All the necessary data is planted inside the file, and the program just pulls out what's there and plugs it into the proper spots.

### Main Task

I think this is my favorite.

```
EXECUTE SUB-TASK setup
EXECUTE SUB-TASK askinput'
EXECUTE SUB-TASK writelib
```

How simple and readable can you get?! Any questions? I didn't think so. Let's move on.

### Setup

In preparing this demo, the programmer has constructed three different text files according to the proper, Madlibs formula. All three of the text files reside in the Program folder with all the other stuff the program needs in order to run. The sample code below is the part of the program which is responsible for asking the user to choose between Hamlet.txt, Catcher.txt and Eleanor.txt. (Perhaps by looking at these file names you can guess which great works of literature we'll be scrambling.) This section of the program also constructs gaggle space for all the literature fragments that, combined with user responses, will constitute the final MadLib.

```
RESIZE CONVERSATION 640,350
PLACE CONVERSATION 0, 100

LOAD STRING "Please enter the name of a file to be read."
WRITE-SCREEN STRING
READ-SCREEN STRING
STORE BOX filename

READ-FILE NUMBER filename
```

*Remember the first line of all these files is a number that corresponds to the number of gaggle items required to handle all of the strings.*

```
STORE BOX numquestions
CONSTRUCT GAGGLE numquestions
STORE BOX questiongaggle
```

```
END SUB-TASK
```

## Askinput

This is the part of the code that reads the top portion of the entries in the file, the ones that contain the requests for verbs, nouns, etc. It also is responsible for storing all the user's responses into the items of the gaggle set up in the previous sub-task.

```
LOAD NUMBER 1
STORE BOX curloopindex                    } top 2 lines make a variable of the gaggle item#

SET GAGGLE-REGISTER questiongaggle

SET LOOP numquestions                     in the case of Hamlet.txt, this box contains 20

@top
READ-FILE STRING filename                 reads the next line down in Hamlet.txt, through line 21

WRITE-SCREEN STRING

READ-SCREEN STRING
STORE-ITEM GAGGLE curloopindex
LOAD BOX curloopindex
ADD NUMBER 1                              these lines relate to the first 2 commands in this task

STORE BOX curloopindex

JUMP LOOP @top

END SUB-TASK
```

## Writelib

This task is responsible for pulling numbers out of the file, matching those numbers with gaggle items containing user responses, and then appending those user responses to fragments of text from the original literature. It's also in charge of knowing when to end. And finally, it gets the user's final results into a file of their own from which they can be reviewed or even printed.

```
SET GAGGLE-REGISTER questiongaggle

LOAD STRING "\\"                          use backslashes as strings to specify carriage returns and line breaks

WRITE-SCREEN STRING                       this inserts two blank lines in the Conversation window between the response requests and the printed MadLib

@top
READ-FILE NUMBER filename
```

```
  COMPARE NUMBER -1                    makes sure program ends at end of
                                       file instead of cycling back to the top
  JUMP = @end
  STORE BOX gagindex

  LOAD-ITEM GAGGLE gagindex
  STORE BOX wordtoinsert

  READ-FILE STRING filename
  APPEND STRING wordtoinsert
  WRITE-SCREEN STRING
  WRITE-FILE STRING "result.txt"
  JUMP ALWAYS @top

@end
  READ-FILE STRING filename
  WRITE-SCREEN STRING
  WRITE-FILE STRING "result.txt"
  LOAD STRING "\\Your 'Bad Lib' has been saved to the file Result.txt.

  END SUB-TASK
```

## What's with All Those Backslashes?

This is something that actually applies to strings in general. When it is printing out strings in the Conversation window, the computer will automatically lump together all the strings it's accumulating to make one paragraph chunk. If you want the computer to "print" a line break or some blank lines between text, you have to tell it to. (By now you should be used to your computer's dim wits when it comes to things like this!) But, of course, there is no form of data called "line break" or "empty space." So, once again, we have to fall back on using the tools at hand to communicate. In this case, we use strings, as the data type, and the backslash character from your keyboard ("\"), *between quotes, of course!* A string containing a single backslash will be translated by FUNdaMENTAL as a line break. (Note that you can use as many backslashes as you need to create the desired amount of space. For example, a string that looks like "\\\" will cause the computer to drop down three line spaces before printing what ever follows.

## Wow! We Said a Mouthful!

There's *a lot* going on in this deceptively simple MadLibs program, as we have seen. It basically rests on this principle: The more thought and complexity you put into your system for arranging text in your text file, the more concise and elegant your program can be. This example illustrates just one way to achieve really good programming style, by creating programs that function as little independent machines, completely detached from any specific data. Stick around with files a little longer, and follow the instructions for setting up

your own MadLibs file according to the formula necessary for running this demo program.

## Let's Make Our Own MadLibs File!

These instructions will demonstrate how to make a MadLib file out of any passage of text you choose. This, of course, will make much more sense if you have a pretty good basic understanding of how the MadLibs program works.

*(Please note: You need to be inside your notepad or word processing application when you try this! You can't create a text file from inside FUNdaMENTAL.)*

The paragraph we're using has been gratuitously stolen from *"Primitivism" in 20th Century Art*, ed. William Rubin. Here it is in its original, "unlibbed" form:

*The crucial influence of the tribal arts– especially those of Africa and Oceana–on modern painters and sculptors has long been recognized. Yet surprisingly, this book is the first comprehensive scholarly treatment of the subject in half a century, and the first ever to illustrate and discuss tribal works collected by vanguard artists. In this visually stunning and intellectually provocative work, nineteen heavily illustrated essays by fifteen scholars confront complex aesthetic, art-historical, and sociological problems posed by this dramatic chapter in the history of modern art.*

Step 1:
Type the paragraph into your notepad or other word processing application. Use Save As from the file menu and save the new text file by navigating through c:\Fundam\Manual\Practice\Demo3 to select the directory for the MadLib program into which you'll be saving this file, and naming the file something descriptive with a ".txt" ending, like "artbore.txt".

Step 2:
Put a "\" at the end of every line. Now the paragraph should start like this:

*The crucial influence of the tribal arts- especially *
*those of Africa and Oceana- on modern painters and *

Step 3:
Pick out all the words you want to put up for grabs, and replace them with numbers. As you go, note the part of speech (e.g. noun, adjective) of each word you replace with a number. The resulting paragraph should now look like this:

*The 1 influence of the 2– especially *
*those of 3 and 4–on modern 5 and *

And your notes should look like this:

1. adjective
2. noun
3. a place
4. another place
5. a group of professionals (e.g. bankers, doctors)

Step 4:
Count how many numbers you have and put this number as the first line in the file, above the first line of the paragraph.

Your text file should now look something like this:

*5*
*The 1 influence of the 2– especially *
*those of 3 and 4–on modern 5 and *

Step 5:
Insert requests for the missing parts of speech between the number at the top of the file and the paragraph underneath.

The file now should look like this:

*5*
*Enter an adjective:*
*Enter a noun:*
*Enter a place:*
*Enter another place:*
*Enter a group of professionals (e.g. bankers, doctors)*
*The 1 influence of the 2–especially *
*those of 3 and 4–on modern 5 and *

Step 6:
Working again with the paragraph itself, remove all new lines from the paragraph like so:

*The 1 influence of the 2–especially \those of 3 and 4– on modern 5 and *

Step 7:
Now, place a new line after every number in your paragraph, so the paragraph portion looks like this:

*The 1*
*influence of the 2*
*– especially \those of 3*
*and 4*
*–on modern 5*
*and *

Step 8:
Cut the number at the end of the line and put it above the line from which it came. Now the whole text file should look like this:

*5*
*Enter an adjective:*
*Enter a noun:*
*Enter a place:*
*Enter another place:*
*Enter a group of professionals (e.g. bankers, doctors)*

*1*
*The*
*2*
*influence of the*
*3*
*–especially \those of*
*4*
*and*
*5*
*–on modern*
*and *

Step 9:
Place a "-1" above the last line of text, if necessary, or as the last line of the file.

You now have a new MadLib file which can be read by your MadLib program!

Notice that the program doesn't have to be changed at all to run with this or any other new text through this MadLibs machine. This is a perfect example of good programming style, which we'll be revisiting in chapter 18.

But first we'll take a little detour into formal logic, FUNdaMENTAL style. Read on to chapter 17 to learn about a new data type called "booleans."

# CHAPTER 17
## True and False with Booleans



Commands Introduced:

## ----- Teacher's Journal -----

I was griping at one of the programmers at the company.

"Okay," I said. "I survived Strings and Loops and even TICKS! But gaggles? GAGGLES? Can we talk? What is the deal with you people and weird names, anyway? And how about 'BOO-Leens'? Huh? What the heck is a Boo-Leen supposed to be? It sounds like someone's skinny, bug-eyed greyhound dog!"

Sarah smiled at me in her most soothing way. She recognizes the symptoms of an acute attack of technophobia. "It's actually just the named for a 19th-century mathematician named George Boole. He formalized logic as we know it. And booleans (pronounced "boo-lee-enz") are all about logic. But now that you mention it, it would make a great name for a dog."

# The Beauty of Logic

As long as it's wires aren't crossed, your computer thrives on logic. That's one of the best things about working with them –or the worst, depending upon your point of view. But in learning a little about programming, even the most steadfast technophobe can begin to see the beauty in logic. And if you happen to be a logic buff already, then this chapter should be a special treat for you.

A boolean is a kind of data with only two possible values: true and false. Since they can only go one of two ways, they are actually pretty easy to use. You can always be sure that if a boolean isn't true, then it's false, and if it isn't false, then it's true. Sounds logical, doesn't it?

## ✔LOAD BOOLEAN

When you use the LOAD BOOLEAN command, it only gives the options of "true" and "false" to load into the AC.

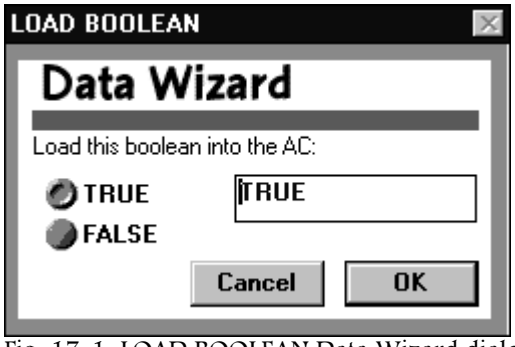


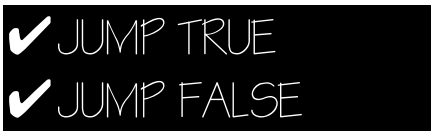

Fig. 17-1 LOAD BOOLEAN Data Wizard dialog.

Just like numbers, booleans can be boxed in such a way that their value is associated with something meaningful. For example, you could make a box called "HasBlueEyes" and store "true" in the box if you do have them:

```
LOAD BOOLEAN true
STORE BOX hasblueeyes
```

and "false" in the box if you don't:

```
LOAD BOOLEAN false
STORE BOX HasBlueEyes
```




We can use JUMP TRUE and JUMP FALSE commands as soon as we load a boolean into the AC. JUMP TRUE jumps to the specified marker when the boolean in the AC is true. I'll bet you can figure out what JUMP FALSE does.

Say we have a box called "I_am_happy" with a boolean in it. We want to write a program where if "I_am_happy" is true, then a smile appears in the Playground window, but if "I_am_happy" is False, then we see a frown. Here's what it could look like:

```
 LOAD BOX I_am_happy
 JUMP TRUE @smile
 JUMP FALSE @frown

@smile
  EXECUTE SUB-TASK smile
  JUMP ALWAYS @end
@frown
  EXECUTE SUB-TASK frown
  JUMP ALWAYS @end
@end
```

## Brushing Up on Formal Logic

Once a boolean is in the AC, it can be manipulated with AND, OR, and NOT commands according to the rules of formal logic. Using the options according to the following table will change the value of the boolean in the AC.

| AC | Operator | Result |
|---|---|---|
| True | And | True |
| True | And | False |
| False | And | False |
| False | And | False |
| True | Or | True |
| True | Or | True |
| False | Or | True |
| False | Or | False |
| True | Not | False |
| False | Not | True |

Illus. 17A Table of logical operators

Just in case your formal logic is rusty, here's the run-down. It helped me when one of the programmers explained about booleans in terms

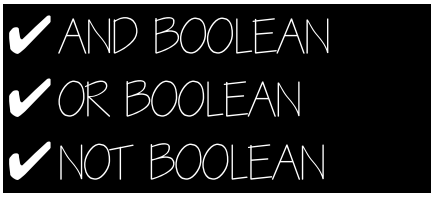
of spoken statements. Let's say it's Wednesday, *and* it's raining outside.

If I say "It's Wednesday *and* it's raining outside" then what I said was true. But if I said "It's Wednesday *and* it's sunny outside" or "It's Thursday *and* it's raining outside" then what I have said is false, right? The first sentence is true because I used two true statements and connected them with the "*and*" operator. But in the other two sentences, I used one true statement and one false statement and connected them with the "*and*" operator, which means that they need to be taken together. As a package deal, they constitute a falsehood.

But, if I say "It's Wednesday *or* it's sunny outside" then what I have said counts as true even though it sounds false because it *is* Wednesday. If I say "It's Thursday *or* it's raining outside" then what I have said was also true, even though it sounds funny, because it *is* raining outside. The "*or*" operator means that only part of what I say must be true for me to avoid telling a lie.

But if I say "It's Thursday *or* it's sunny outside" then this is false, since both options given are false.

Put another way, the first two sentences are true because I used at least one true statement and connected the two statements with the "*or*" operator. The last sentence is false because I used two false statements and connected them with the *or* operator.




Using any of the three commands above performs the indicated logical operation on the true/false value in the AC, with respect to another true/false value that you specify. When you use any of these commands, the Data Wizard will bring up a little dialog which allows you to click a radio button for "True," "False," or "Box." (Of course, if you choose a box here, it will be one in which you've already stashed a boolean value of "true" or "false." True? True!)

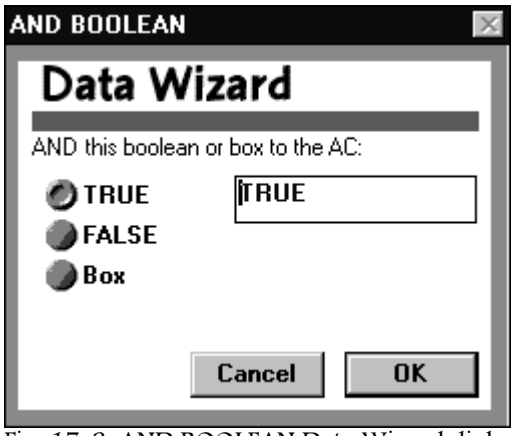


Fig. 17-2 AND BOOLEAN Data Wizard dialog.

## "Okaaay..."

It may be difficult now for you to see how all this ties into making really cool programs. And I must admit that I've never met a programmer who said that booleans were *absolutely necessary* for programming.

But one programmer I know likes to make elaborate games. He told me he recently used several boolean commands to make his program check to see if the action hero was tall enough, strong enough *and* rich enough to receive a reward toward victory. "It's kind of like making your own, customized C-bit," he said.

I looked at him with widening eyes and said, "Oooooh, I think I'm beginning to see."

For him, booleans were just another way to have control and flexibility over how comparisons and possibilities helped him reach his program goals. And even if you never use them to reach your own program goals, booleans are kind of fun to play around with in their own right–especially if you happen to really like formal logic!

## Flags

My programming friend was using ooleans as "flags." A flag is when you take something the computer understands, like a number, or in this case, a boolean, and use a box to attach it to something the computer doesn't understand, for use in a program. For example, the computer can't deal with pairs of possibilities like "happy or sad," or "strong or weak." But it does understand "true" or "false." So for the sake of your program, you could *call* the "sad" possibility "false,"and the "happy" possibility "true." The computer doesn't have to know what "true" and "false" mean in any given instance–only you do.

An example will help, so let's write one. In the next exercise, we're going to use the values of "true" and "false" as a way to tell if a light bulb graphic is morphed "on" or "off."

## LET'S PROGRAM!

1. Launch FUNdaMENTAL, and open Progrm10.fmp. At the FM Welcome window, click on Open Existing Programs, and then navigate through the following directories: c:\fundam/manual/practice/progrm10

2. Look at the code in the main task, and test-run the program. As you can see, a boolean value of "false" has been stored in a global box called "onOffFlag." Also, a "bulb" object has been constructed, placed, and shown, and a box called "theBulb" has been defined as a global box. You can see the bulb when you test-run the program. The "false"-in-a-box, sets up the flag we'll be needing in the sub-task.

3. Stop and think for a minute about using booleans to make an "on/off" click-task. It seems simple at first, but it's actually a little tricky to make a click-task that does one thing the first time you click on it (click "on!"), and another thing the next (click "off!").

4. Create a new sub-task called "bulbclic." Use the Window menu and select the Program window. Click the New button, and use the file name entry field to enter the name "bulbclic.fmt". Click OK, and you'll automatically wind up in the blank task window for the new sub-task, "bulbclic.fmt".

5. Use the command, LOAD BOX, and select the global box, "onOffFlag."We know that the first time "bulb" is clicked upon, this box will contain the boolean value of "false," because that's what we stored in the main task. That "false" is what the computer uses to know that the bulb appears in its "off" form, and should be morphed to "on" when clicked.

6. Use the command JUMP TRUE and define a marker called "@turnoff." Here's where you have to think of how your program runs through time. The first click will need to turn the bulb on, but the next click should turn it off. This JUMP command covers the second possibility, and tells the computer to go follow "off" instructions. When you use this command, you can just type the marker text into the entry-field in the Data Wizard dialog. We'll put the marker in the Instruction list after we write the "on" code.

7. Use LOAD BOX and select "theBulb" from the box list. Now the AC contains the bulb and is ready to morph some light on the subject.

8. Use MORPH OBJECT and select "bulbon.bmp" from the list of available graphics.

9. Use LOAD BOOLEAN TRUE. This command will serve as a signal "flag" to the computer that the bulb is now in its "on" state, and the next click should turn it off, instead of turning it on again.

10. Use JUMP ALWAYS and define a marker called "@end." I bet you thought we were about to use a STORE BOX command, didn't you? Well, we are, but we're going to do it at the end of the task, so that whichever boolean happens to be in the AC at the time will get stored in the "onOffFlag" box and be ready to go for the next click. We'll be putting the command STORE BOX onOffFlag under the "@end" flag at the end of the task.

11. Now place your "@turnOff" marker in the code. This marks the code for the JUMP command we placed near the top of the task. Click on the radio button labeled "marker" above the Instruction list, and type the text after the "@" which appears in the entry field.

12. Use LOAD BOX and MORPH OBJECT, to get the bulb object into the AC, and morph it to the "off" form. You'll be looking for a graphic called "bulboff.bmp" in the available graphics list in the MORPH OBJECT Data Wizard dialog.

    Use LOAD BOOLEAN FALSE, to switch the flag and signal the computer that the next click should turn the bulb on again.

13. Place your "@end" marker in the code. This marks the place for the JUMP ALWAYS that we put just before the "turnOff" code. Click the "marker" radio button about the Instruction list, and type the text in the entry field.

14. Use STORE BOXonOffFlag. Now, whatever is in the AC, "true" or "false," will be stored in the "onOffFlag" box, ready for the next click.

15. Use END SUB-TASK.

16. Test-run the program, and try clicking on the light bulb to see your click-task work.

## Programming Tips and Tricks

Once you get a chance to see how booleans work as "flags" in this program, you may notice that this is not the only possible way to make flags. For example, you could achieve the same effect with number data loaded into the "onOffFlag" box. Can you see how it would work and what changes you'd have to make in the code?

# CHAPTER 18
## "Everything Is Trees": Elements of Programming Style

## ----- Teacher's Journal -----

I was sitting with one of the programmers, trying to work out the solution to a teaching problem. I was concerned that in attempting to make the material accessible to my young students, I was limiting their sense of the true nature and potential of programming.

"They know how to use the environment, and their vocabulary of commands gets bigger every day," I said. "But how can I communicate to them what programming really IS…how can I teach them at least to appreciate the elements of style and efficiency that expert programmers incorporate into their work—stuff I can recognize in the programs you guys write, even if I can't write them that way myself?"

Sam began doodling on his yellow pad and said "Well, of course learning the art of decomposition is a whole college career in itself. But basically all good programs are shaped pretty much the same way. They look like this…" He added a few more strokes to his drawing.

"It boils down to something pretty simple," he said, handing me the picture showing a collection of loops branching symmetrically away from a central line. "Everything is trees."

## The Beauty of It

The more you learn about programming, the more you'll see that Sam is right. Really good programs are indeed "trees," in which sub-tasks relate to the main task (and to each other) the way branches of trees relate to the trunk, or the way coral divides away from its anchor point.

But that's just the root of it.

At the heart of advance programming lies the ability to visualize a system in four dimensions. That's right: Three dimensions of space, and then toss in time.

As you advance in your programming, you'll learn to think of every task as a minimachine that's doing something. Each one is connected to the other tasks through EXECUTE SUB-TASK and EXECUTE PROCESS-TASK commands to make one big machine that is your program. Once the machine is assembled in your mind, add the data. Think of the data as flowing from one minimachine to another through the connections by way of parameters (information and specifications "passed in" from one task to another.) Now you have to imagine the machine as it runs through time. If you can do that, you're the J.S. Bach of programming. Luckily, we don't have to think of it all at once…at least not if we use good programming style!

Throughout this manual, we've dealt with various skills and concepts that will help you see your programs in this way, and manage them, once you do. Let's revisit some of those key ideas now.

## Decomposition

"Decomposition" is the art of breaking things down into manageable, component parts. If you want to write a program to keep track of your checkbook, for example, think about what is entailed on the top level first. Keeping track of checks involves recording deposits and withdrawals. Each of these actions should be a sub-task of the main program. You've already broken the program in half. The secret of decomposition is that it's easier to write two halves than a whole.

Now break down the problem of handling a withdrawal. That consists of finding out how much was withdrawn, subtracting that amount, and recording the new balance. Aha! There are three sub-tasks in withdrawal. Just like magic, the problems left to solve are even smaller. Keep doing this until each task you have is small enough that you can easily write it.

This kind of top-down design is the key to happy programming. If you take the time to break each task into smaller tasks, until the commands in a task are obvious, you'll save yourself much debugging agony later. Trying to shove all your commands into the

main task is like eating without chewing: You and your program will probably choke.

## Modularity

Another thing to keep in mind when dividing your program into tasks is "modularity." This is a nice word for breaking your program into conceptual chunks. Good modularity is what results when you think of your boxes as variables, and make sure that each sub-task is equipped with local boxes so that it can be easily read and understood on its own. This is good for a variety of reasons, but it's especially helpful for program troubleshooting. If your program is divided into good, modular chunks, then problems that can (and will!) arise are more easily traced and rooted out.

Think about it this way: Each task should have one specific function. Let's say your program involves finding the square root of some numbers (rounding down to the lowest integer). This only takes a dozen or so commands, so you might be tempted to just put these lines into whatever task needs them. Don't give in to temptation, because there are many reasons for making a separate task that takes a number in a box, finds the square root, then puts that number back in the original box.

The first reason is that once you write that task and get it to work, you can close it and never worry about it again. If you had left it in with all the other commands, and something didn't work right, you would have to examine every line to find the mistake. Modular writing of tasks allows you to concentrate on and test individual parts of your program without worrying about what's happening on other levels.

The other major advantage of writing modular tasks is the time it saves you rewriting commands, because it allows for…

## Reusability

In actuality, we've seen that good programs are more like Tinker Toy trees than the botanical kind. The reason is that advanced programs often contain sub-tasks, or "branches," that are interchangeable in that they can be used and reused in a variety of different programs. If you've written that square root task well, you can pick it up and plop it down in any other program and it will work fine. How do you write a task so that it can be reused? Avoid relying on global boxes. Global boxes that are in one program aren't necessarily in another, and if your task depends on their existence, it won't work without them. For this reason, use local boxes wherever possible.

Reusability also applies to lines of code within a single task. Let's say you use the RANDOM NUMBER command to randomly pick one of three frogs to turn into a prince. You could

jump to a separate section for each number, load the object and change it, or you could jump to a separate section, load the object, then jump to a common section where the object in the AC is altered. This saves you from having to write the each line three times.

## Readability and Naming

Program "readability" is a primary ingredient of good programming style. A lot of things go into making a program readable, from one programming session to the next, and between different programmers. We've already seen how good decomposition, and modularity, and reusability can help. Another extremely important skill of programming in any language is the use of good, "descriptive names."

If you name all the tasks in your program "task1," "task2," "task3," and your boxes are "box1," "box2," "box3," you'll quickly lose track of what's going on. Name everything according to its purpose. If a box holds a Gaggle of sharks, call it "GaggleOfSharks," or "sharkgag," for short. If a label marks the spot in your program to go to when the ball is done bouncing across the screen, name the label "@ball_done_bouncing," or something similar. You'll be amazed how much of a difference this makes in your ability to follow what is happening in your program and see where the mistakes are.

You don't have to be an expert programmer to appreciate the elements of style in programming. The more experience you have in programming, the more you will recognize that all of this technology and cyberspeak is really just another way to give form to the order that characterizes critical thinking, and that mimics patterns found throughout the natural world.

# UNIT 3
## Highlights

## COMMANDS

CONSTRUCT GAGGLE
SET GAGGLE-REGISTER
STORE-ITEM GAGGLE
LOAD-ITEM GAGGLE
DESTROY GAGGLE
COUNT GAGGLE
GET LOOP
WRITE-FILE STRING
WRITE-FILE NUMBER
READ-FILE STRING
READ-FILE STRING
REWIND FILE
LOAD BOOLEAN
AND BOOLEAN
OR BOOLEAN
NOT BOOLEAN
JUMP TRUE
JUMP FALSE

## PROGRAMMING STYLE

Decomposition
Modularity
Reusability
Readability and Naming

## INSIDE THE COMPUTER

Gaggle Register

# APPENDIX A
## Educator's Grab-Bag

Section 1A: The FUNdaMENTAL Learning Process, in Principle and in Practice

Section 2A: Try This in Your Class Tomorrow!

Section 3A: "What's So Great About Programming?": Talking to Parents, Administrators, and Older Students

This section is designed to give you extra support as you prepare to bring FUNdaMENTAL into your classroom or computer lab.

In section 1A you'll find a description of the FUNdaMENTAL learning process as we've observed it during field testing. Each learning principle is presented with corresponding implications for optimal teaching practices.

In section 2A you'll find a series of tried and true lesson designs for a variety of online and offline activities which support the learning of FUNdaMENTAL.

In section 3A you'll find suggestions for talking to parents, administrators, and older students about the benefits of FUNdaMENTAL, and programming in general. Included are guidelines and curriculum standards from a variety of educational frameworks.

# SECTION 1A

## The FUNdaMENTAL Learning Process, In Principle And In Practice

What follows is a series of learning principles that we've derived from our observations during the field-testing of FUNdaMENTAL. Each one describes a general feature of the typical beginning learner's development and state of mind in learning programming. Each learning principle is followed by general implications for classroom practice which, when applicable, can help you to get the most out of your FUNdaMENTAL curriculum.

## IN PRINCIPLE AND IN PRACTICE...

✔ FUNdaMENTAL learners need to reinvent their customarily passive users' role into one that is dynamic and creative.

In Practice:
It helps to hold many discussions about favorite software titles, and raise questions about how software is made in the first place. Specific references to programmers and software companies help to demystify the phenomenon of computer technology.

✔ FUNdaMENTAL learners almost always find animation and graphics to be the most engaging and motivating place to begin learning about programming.

In Practice:
Start with objects. This is not as obvious as it seems to someone who learned this way. Strings actually make for easier programming because you don't have to stock any libraries or design any blueprints. You can just jump right in. Trouble is, in our experience most students (especially younger ones) just can't get too excited about strings until they come to understand something about programming through the medium they know best: graphics!

✔ When starting out with animation, FUNdaMENTAL learners need to be sheltered from peripheral interface challenges so they can focus on the process of programming itself.

In Practice:
Use premade program starters and program frames, like the ones accompanying this manual, for beginning classroom practice and computer lab time. Try to ensure that inexperienced students don't confront interface difficulties that overshadow the programming challenges at the center of your lesson.

✔ Depending upon their prior experience and goals, FUNdaMENTAL learners will respond to the learning process in various, and somewhat unpredictable, ways. They will benefit most from lessons designed around their specific needs and interests.

In Practice:
Because programming is much better described as a "process" than as a "skill," we have found it beneficial to teach FUNdaMENTAL in a "workshop" atmosphere. Although goal setting and planning are essential in any educational endeavor, teachers of FM need to make the time, on a regular basis, to observe students at work. Lesson plans should incorporate activities that respond to specific student needs and interests.

✔ FUNdaMENTAL learners need plenty of time to "spread out" and explore at each new level of understanding. Even the simplest of program starters or classroom activities can trigger a challenging creative effort. The benefits of "horizontal" curriculum can vary, depending upon both the needs of the students and the particular topic of study within the discipline of programming.

In Practice:
Again, a workshop approach works well here. Be willing to individualize. Plan activities in a goal-based menu format, and know that different students will end up in different places at the end of any given lab time. If you use cooperative learning structures in your teaching, make sure to plan for both heterogeneous and homogeneous groupings. Pairing students according to common interest with respect to program outcomes (e.g. "We both want to make a boat race program.") is one good way to make sure all students are invested in the process, regardless of differences in their programming abilities.

✔ FUNdaMENTAL learners will be continually reinventing what they know about programming. Learned concepts will remain in flux as new experiences add depth and complexity to the learner's understanding.

In Practice:
In teaching something as complex as programming, one of the greatest challenges is finding the right place to begin. Presenting material in a spiraling format can go a long way toward solving the problem. Know that it's okay to present part of the truth– just enough to initiate and support your students' hands-on experience with programming. When you spiral back around to revisit concepts again, your students will be in a better position to absorb more of the truth about difficult concepts. Help the students recognize, predict, and welcome the changes in their thinking as they progress with FM.

✔ FUNdaMENTAL learners may be inclined to hold onto metaphors and comparisons in order to resolve their early uncertainties about programming. Although sometimes helpful for building beginning concepts and putting learners at ease, such comparisons can often be limiting, and may work against a full understanding of the ideas they initially seemed to illuminate.

In Practice:
Be frugal and judicious in your use of metaphor when teaching FUNdaMENTAL. When I started out, I found myself wanting to compare different aspects of programming to any number or wild things, from making potato prints (with respect to the AC), to making omelets in my mother's kitchen. Although the prospect of clearing things up may be just as tempting to you as it is to your students, don't give in to the impulse to make broad comparisons without giving it a lot of thought first. Condone uncertainty as part of the process. Endure the occasional perplexed grimace and blank look. Stick with "the computer" as your primary point of reference, and your students will almost surely benefit from a fuller understanding later on.

✔ FUNdaMENTAL learners will learn more quickly, and program more effectively and efficiently, if they understand the computer's inner workings.

In Practice:
The one metaphor that does consistently pay off over time is the metaphor that explores the question "What's going on inside the computer." To those of us with little or no experience in programming, it may be hard at first to sense the presence of a metaphor here. But when you really think about it, talking about "putting things in" the AC, or "setting" the C-bit or the Loop Register

doesn't really get at what's going on inside the computer. It may comfort you to know that many professional programmers don't actually grasp how the computer chip executes commands. It's enough to help students form a strong sense of a physical model to represent this mystery in order to help them become powerful programmers.

✔ FUNdaMENTAL learners will best learn how commands affect the "inside of the computer" by using commands to write programs.

In Practice:
Don't try to force an early understanding of the computer model. Nor is it necessary to hold back from introducing new commands until students master all the concepts associated with the ones in previous lessons. Keep the commands themselves at the center of your early lesson planning. Hands-on programming experience will allow the students to make more and more use of the model both for better understanding and for program troubleshooting.

✔ Learning FUNdaMENTAL can be very frustrating.

In Practice:
Teaching FUNdaMENTAL can be very frustrating.

✔ Learning FUNdaMENTAL can be very, very rewarding.

In Practice:
Teaching FUNdaMENTAL can be very, very, VERY rewarding.

# SECTION 2A
## Try This In Your Class Tomorrow!

### Tried And True

- How to Make a Peanut Butter Cracker
- FM Coordinate Battleship
- What's Wrong with This Code?
- Command Lineup
- Prediction Activities
- Program Planning Sessions
- Company Meetings

### More Ideas

- Cue Card Code
- Reflection/Prediction/Planning Journals
- Programming Across the Curriculum

Although the tutorial portion of this book may serve as the foundation for your online lesson planning, there is a whole collection of other learning experiences that can enrich the program you build around FUNdaMENTAL. The majority of these are offline activities that should be done, if possible, with the class assembled away from the computers (or at least with backs turned).

The first group of activities we have personally tested in the classroom and have been found to be successful. The majority of our testing grounds were at the junior high school level, but the tools for collaboration and reflection presented here will be helpful for students of all ages.

The second group of ideas are ones we have not yet had the chance to test ourselves, but they have been suggested to us by others who have brought FUNdaMENTAL into their classrooms.

# Tried and True

## How to Make a Peanut Butter Cracker

Who: Great for all students new to programming/whole group or rotating clusters
Why: Helps build foundation for understanding task decomposition and the programming process
When: Third or fourth class session, after students have had some hands-on programming experience
How: At a place away from the computers and visible to all students, display the following items:

-small jar of peanut butter, unopened (and sealed if possible)
-package of crackers, unopened
-knife
-plate
(Feel free of course to add ingredients of your own, but keep it simple. You'll be amazed how involved things can get!)

Also have handy:
-sentence strips, or other large strips of paper
-2 large felt-tip pens

Tell the students that you would like to treat them all to a snack. The only thing they need to do is to tell you how to make a peanut butter cracker. Appoint two recorders to keep track of the instructions on the sentence strips, so you can examine them later.

Stand in front of the table of ingredients with your hands at your sides. Smile pleasantly and raise your eyebrows in a docile fashion. Before long, some hungry soul will start to call out instructions.

Here's the fun part (and the challenge!): You have to follow their instructions exactly. That means, you don't do anything that they don't tell you to do, and you do everything they tell you precisely as they say it.

Feel free to "crash"… that is, to decline to follow an instruction that is incomplete or out of sequence. For example, my students almost always start out with the instruction "Open the peanut butter!" (In your mind, visualize the FM translation: OPEN peanutbutter). That's your cue to say, "Peanut butter? What peanut butter? I'm not holding any peanut butter." (Read, "Wrong AC type.")

After that, they're off and running. It's always great, aggravating fun for the group to collectively deconstruct this deceptively simple task.

"Okay! Pick up the jar!" ( I pick up the jar.)
"Now open it!" ( I like to start banging the jar against the table top, or shout "Open, Jar!" in response to this one)
"No, take off the lid!"
("Off, Lid!)
"No, put your hand on the lid…", "The PALM of your hand…" someone else chimes in. (Now they're getting the hang of it!)

The best part, of course, is when, after ten minutes of arduous group programming, there's *one* lonely peanut buttered cracker in the middle of the plate to show for it.

"Do it again!" some of my students shout.

"Repeat!" command others, trying for a slightly more technical tone.

"What, all of it?" I answer, starting to go through the motions of breaking the seal on the already open jar.

The first time I tried this lesson, I discovered inadvertently that it was a great way to set my students up to learn about SET LOOP/JUMP LOOP. Even the sometimes tricky issue of marker placement comes up as students review their list of instructions to identify the parts that need to be repeated.

# FM Coordinate Battleship

Who: Good for younger students who may not yet have experience with coordinate grids/Whole group or partners

Why: After introducing MOVE OBJECT, you may find some students still unable to focus on the programming task because they are unclear on how to think about object placement and movement in terms of coordinates. Even with the Show Grid button on the Toolbar, you may find that some students benefit from some offline practice. What better reason to learn/review this basic math skill?

When: Try this lesson after students have had at least one experience programming with MOVE OBJECT. That way they'll be sure to connect the lesson to what they're doing in FM.

How: There are various approaches to offline activities that give students extra practice moving things around on a grid. To do them you should have at hand:

- -A xeroxed copy of the coordinate grid for every pair of students; or one large grid on butcher paper
- - A moveable Object: cut-out graphics, beans, or coins...anything will do
- - If you're working whole class, you may want a large number of strips with various x/y coordinate pairs printed out on them for volunteers to choose randomly

If you're working with the whole class, you can have different volunteers come up to the large grid and move an object according to the coordinate "input" on a slip of paper that another volunteer draws from a hat.

Or, pairs of students can sit with grids on opposite sides of a barrier so that they can't see each others' work. Starting with their object at (0,0), they can take turns calling out MOVE commands which both partners follow simultaneously. After six turns, the barrier is removed, and partners check to see if their Objects ended up in the same place. Remind students to record their moves on a separate sheet of paper, so they can retrace their steps if they come out in different spots.

And of course, pairs of students can always play Coordinate Battleship. For those of you who never played this game as a child, here's how it goes. Once again, partners need to sit with a barrier between them. Each player has a grid on which s/he places several objects at various different coordinates. Players take turns calling out coordinates, trying to be the first to find all of their opponents' objects. This helps students familiarize themselves with the territory. It may even lead to inspiration for a future programming project...

## What's Wrong With This Code?

Who: All students of FUNdaMENTAL/Whole class, small groups, or pairs

Why: Aside from being a great critical thinking activity in its own right, this lesson is extremely effective for addressing recurring problems and troublesome issues that you observe during your students' lab time.

When: Any time and all the time. You can use this activity as a warm-up, a wrap-up, or a mid-lesson intervention. Begin using it as soon as the students have enough programming experience to start making mistakes.

How: The activity title says it all. This simple, powerful lesson consists of students analyzing a piece of faulty code to find the flaw. To do this activity, you'll need...

-a stretch of faulty code printed on individual sheets, butcher paper, or an overhead transparency

You can use any faulty code at the students' level, but the most effective ones come straight from your observations of your students' work.

When you first begin, it helps to give students a short description of what happens when you try to run the faulty program ("This is supposed to be a jumping kangaroo, but when you push Play, it looks like he's trying to fly to the moon; he just goes up, up, UP!"). As they progress and become more experienced, ask them to tell you the nature of the program "crash."

## Command Lineup

Who: All FM students, but more often those having little or no programming experience/Whole class or small clusters
Why: To get students thinking about task decomposition and logical sequencing
When: As soon as students have had some hands-on experience with FM, and as needed to help students tease out the sequence of things in their programming.
How: This activity consists of having students physically manipulate a set of commands that are printed on individual strips in order to…well, put them in order! To do this activity, you'll need:

-commands (the larger the better) on strips

Although partners can do this with small strips at individual work stations, I always like to make this activity a group affair. The commands are passed out, one to a customer, and everyone has to physically find his/her place in a line-up. This approach invites comments from the peanut gallery, and gets everyone involved in an active, nonthreatening activity.

# Prediction Activities

Who: All FUNdaMENTAL students regardless of ability and experience/Whole class, small clusters

Why: Because planning ahead and analyzing outcomes is what it's all about

When: After about the first 10 minutes of the first FM class, it's safe, and educationally profitable, to start using prediction activities; anytime, all the time, warm-up, wrap-up or mid-class intervention

How: Prediction activities can go both ways. You can either show students the program output and ask them to predict what commands they'll see in the Task list, or you can show them a stretch of code and ask them to predict what the program will look like. To do this you'll need:

-sample programs

Students can either work out loud together or individually in writing to make their predictions. You can bring small groups of students around individual computers or if you're lucky enough to have an overhead projector connected to one of the computers in the room, you can do this as a whole class. One way for the whole group to predict program output (that is, look at commands and try to imagine what they'll do when the program is run), even if you don't have one of those fancy overheads, is to print the program up on butcher paper. After the class has had a chance to voice ideas, have small groups of students take turns running the program on a designated computer during lab time.

## Program Planning Sessions

Who: Beginning intermediate FM students who are ready to start running the whole show/Individuals or pairs

Why: Although program plans will inevitably grow and change throughout the creative process, it's essential that students form the habit of envisioning their program before diving in to start building it. This helps condition them toward proactive, "decompositional" thinking, and also yields a documented plan which can help them avoid being overwhelmed at the outset.

When: At the start of independent programming projects

How: Set aside a good portion of one class period for offline planning. The first time you present this activity, younger and inexperienced students will find it quite challenging. You should demonstrate by talking through the process of filling out a Program Planning Sheet. To do this you'll need (you guessed it!):

- sample Program Planning Sheet, printed up on butcher paper or an overhead transparency
- one copy of the same sheet for every student or pair of students who will be writing a program from scratch

We've included a sample Program Planning Sheet, but please use this only as a starting point. All planning sheets should contain space to write about, or, better yet, sketch the desired outcome. They should also include a place to brainstorm key commands. Beyond that, it will be up to you to guide their planning depending upon what's going on in class at the time you start the project.

For example, at the time I made the Planning Sheet included here, my students were still struggling to distinguish between objects and graphics. They still didn't quite grasp the "flip-book" phenomenon which requires that a single object (like a diver or a prize fighter) be represented with several graphics showing the figure in different positions. In order to cue their thinking in this direction, I designed the Planning Sheet with the difference between graphics and object in primary focus.

As your students advance, the issues important to their planning will change. Your planning sheets should change along with them.

# Program Planning Sheet

Name:
Date:
Period:

Proposed Program Title:

What will your finished program look like?…sound like?…do? (Describe and make a sketch on the back of the sheet):

Name the object types you will have in your program, and tell how many of each type you'll be using:
-Object 1: #:
-Object2 #:
-Object3 #:

Name the graphics you'll need to represent each object as it moves through your program.
Object 1:
-Graphic1:
-Graphic2:
-Graphic3:

Object 2:
-Graphic1:
-Graphic2:
-Graphic3:

Object3:
-Graphic1:
-Graphic2:
-Graphic3:

List the first 5 to 10 commands that you think you will need to start this program:

List any questions you have about building your program, or problems you anticipate facing as you work:

## Company Meetings

Who: Junior high and high school FM students/Whole class
Why: Helps demystify programming process by elevating student efforts in their own eyes; provides career awareness; models positive practices from private industry. (When I went to work for KartoffelSoft, I was immediately struck with the productive and communal feeling at the company meetings. Company meetings accomplish three functions at once: community building , troubleshooting, and goal-setting.

I've always had the same three goals in mind in planning meetings in the classroom. And yet, somehow, they never quite seemed to jell in the same way they did in the company loft.

It occurred to me that perhaps my students never really understood that when we were "just sitting around in a circle talking," we were actually modeling something that goes on all the time in the "real world" (What, our classrooms are in some kind of fairy land?); and it's serious business.

At the outset of the class, you may want to invite students to come up with a company name. And once they get a sense of what programming is and how it works, it's good to set a large-scale company goal, such as putting on a computer fair for a younger class, or making a collaborative exhibit around a particular theme…Once your class has a new identity and a collaborative sense of purpose, your company meetings can really take off.)

When: As soon as students begin programming
How: To make this activity part of your enterprise's routine you need:

-a nice, strong voice to holler "Com-panyyyy Meeeeeting!" in a way that's loud and persuasive enough to get those kids away from the computers
-an optional secondary signal (I'm partial to a train whistle myself...) that has been pre-established as the signal to take those hands off the computers and listen up
-a clear and consistent agenda that follows generally the same pattern each time

Students should begin with a check-in to share with the group what they've been working on since the last meeting. This should give rise to issues for celebration and troubleshooting. Make sure to end each meeting with goal setting so that all of your students are clear on where they are going when they pick their up work once again. If you're working with a large group, certain aspects of the agenda can

be handled within small clusters and then summarized for the whole group at another time.

# More Ideas

## Cue Card Code

Take a short sequence of code and write it out on a large sheet of paper. Have large cut-outs representing the graphics and strings of a program, and hand those out to individual volunteers. Assign one volunteer to be the Central Processing Unit. Have the students act out the code to show what it would look like when it was run, with the designated CPU acting as choreographer/conductor.

I'm not even sure if this would work…What do you think?

## Prediction/Planning/Reflection Journals

With all of the predicting and planning and trouble-describing/shooting going on it would be very useful to include regular journal writing in the FM workshop. Depending upon the literacy needs of the students and their level of experience with technology, you may decide to have set forms or just notebook paper.

Journals could form an integral part of the Company Meeting cycle, with students reflecting on paper before or even during the meetings. Aside from providing students with a valuable opportunity to use writing as a tool for math and technology, journals would be a great help to us when it came time to assess student progress in the class.

## Programming Across the Curriculum

If you're teaching in A self-contained, multiple-subject setting, then I'm sure this has occurred to you, as well. As soon as I learned about FUNdaMENTAL, I started envisioning what programming could do for a classroom community. I figured that, by about February, we could count it among our standard tools for accomplishing academic and extracurricular goals.

I imagined students in committees making programs for practicing spelling or math facts, with the contents tailor-made to fit the current curriculum.

I imagined them doing the same thing for younger tutoring buddies in lower grades.

I imagined interactive book reports on books we'd just finished.

The whole concept of the shoe box diorama, or the painted mural, took on new dimensions, as I imagined how they could translate into dynamic programs to culminate social studies or science units.

And why not have students custom-design programs to keep track of some of the community business that is so central to the fabric of classroom life? I could assign students to create programs that were exactly tailored to the specifics of our classroom jobs, homework routines, and student information.

Even if you teach in a single-subject, computer lab setting, I would like to encourage you to connect with teachers in other curriculum areas to get ideas for how your students might reach out to the school, and wider communities, as their programming skills increase. Is there a particular academic skill or theme you could emphasize in structuring programming assignments? Is there a group of younger students who could benefit from some custom-made programs? Are there community service or charity groups that might like to sponsor a computer fair, with programs that illuminate certain issues or promote a certain cause?

We're eager to hear from any of you who read this and decide to take this challenge. Visit our Website at www.kartoffelsoft.com to share your ideas, and to find out what others are up to in exploring the outer limits of FUNdaMENTAL's potential.

# SECTION 3A

## "WHAT'S SO GREAT ABOUT PROGRAMMING?"

---

## Talking to Administrators, Parents, and Older Students about the FUNdaMENTAL Benefits of Programming

*If only a handful of your students may actually choose programming as a career, why should you spend so much time and energy teaching programming to everyone?*

At this point, we're confident that we don't need to answer this question for your benefit! But we're well aware that in your educational practice, it may be necessary for you to answer the same question, put to you by administrators, parents, and maybe even older students. In this section we present the most compelling reasons why programming is good for all students, in order to help you communicate with others about your decision to use FUNdaMENTAL in your teaching.

## What Is "Critical Thinking," Anyway?

Although many outside the field of computer science may think of programming as among the most specialized of skills, in truth, it embodies a process that is central to almost every higher-level academic endeavor: *critical thinking*..

It's a term that educators and parents alike are so used to hearing, that for some it may have lost its meaning. But critical thinking is at the heart of what's good about programming, so it's a good idea to clarify what it means to you.

The following provides one definition. "Critical thinking skills [include the ability to] define and clarify problems; judge information related to a problem; solve problems and draw conclusions." Certainly all three of these skills are directly developed and enhanced when a student becomes involved in programming a computer. And please note that this particular definition of critical thinking comes straight from the California State Framework…*in History and Social Science!*

## Higher-Order Thinking Skills

Closer to home, the California Mathematics framework (citing L.B. Resnick, 1987) lists the following features as recognizable in "higher-order thinking":

- ✔ Higher-order thinking is non-algorithmic. That is, the path of action is not fully specified in advance.
- ✔ Higher-order thinking tends to be complex. The total path is not visible (mentally speaking) from any single vantage point.
- ✔ Higher-order thinking often yields multiple solutions, each with costs and benefits, rather than unique solutions.

- ✔ Higher-order thinking involves nuanced judgment and interpretation.
- ✔ Higher-order thinking involves the application of multiple criteria, which sometimes conflict with one another.
- ✔ Higher-order thinking often involves uncertainty. Not everything that bears on the task at hand is known.
- ✔ Higher-order thinking involves self-regulation of the thinking process. We do not recognize higher order thinking in an individual when someone else "calls the plays" at every step.
- ✔ Higher-order thinking involves imposing meaning, finding structure in apparent disorder.
- ✔ Higher-order thinking is effortful. There is considerable mental work involved in the kinds of elaboration and judgment required. (Mathematics Framework for California Public Schools, Cal. State Dept. of Education, 1992, p.21)

This, too, reads like a FUNdaMENTAL checklist of the qualities present in your programming lessons. Check out your own local mission statements, model standards, and state frameworks, in all areas of the curriculum. You're sure to find equally compelling arguments for using programming as a means of promoting critical thinking in your students.

## Meaningful Technology Education

In addition to the general desire to develop students' critical-thinking skills, we are experiencing an ever more urgent need to give students meaningful experiences with technology. Again, this is something we hear about constantly as we approach the 21st century. So, once again, you will have to stop and ask yourself how you define truly meaningful technology education.

Just plunking down computers in the classrooms certainly doesn't make the grade. Nor does the use of software which is essentially glorified, electronic flash cards or video.

In his book *Mindstorms: Children, Computers and Powerful Ideas* (New York: Basic Books, Inc., 1980), the mathematician Seymour Papert observes that, all too often, computers are used to program the child, instead of the other way around. He specifically promotes programming as a learning experience that is "more active and self-directed. In particular, the knowledge is acquired for a recognizable personal purpose. [The learner] does something with it. The new knowledge is a source of power, and is experienced as such from the moment it begins to form in the [learner's] mind." (p. 5)

## Learner-Centered Software

In outlining educational practices that promote "mathematically powerful" students, the California Mathematics Framework strikes a similar chord with respect to the use of technology. Citing *Beyond Drill and Practice: Expanding the Computer Mainstream* (Russell, et al1989), the framework identifies the following characteristics in "learner-centered" software:

- ♦ Learner-centered software offers students choices in selecting the goal of the activity, the strategies to reach the goal, or both.

- ✔ FUNdaMENTAL offers both!

- ♦ Learner-centered software provides feedback that is informational rather than judgmental.

- ✔ In FUNdaMENTAL, there are no Whoopsie Whistles of golden biscuits. All of the explicit feedback is informational and, in the end, your program either runs or it doesn't!

- ♦ Learner-centered software allows, emphasizes, or encourages prediction and successive approximation.

- ✔ FUNdaMENTAL is all about making predictions, and revisions to reach ever more challenging programming goals.

- ♦ Learner-centered software encourages learning within a meaningful context for students, building on students' intrinsic motivation.

- ✔ Since FUNdaMENTAL is truly interactive, it's the students' own visions and goals that guide, and reward, their efforts. The best reward is a program that runs, entertains and amazes!

As you can see, FUNdaMENTAL was designed with each one of these qualities in mind. Regardless of their particular career goals, students can only benefit from working with software that is truly learner-centered according to the above definition.

But you don't need fancy academic definitions to tell you when something you do with your students is learner-centered. For me, the meaning of learner-centered was summed up the day that LaToya wrote her diver program. I didn't need to check my district standards to know the value of her learning experience. It was simple: the bell rang, class was over, and LaToya wouldn't leave!

## When Kids Won't Leave The Classroom

But you don't need fancy, academic definitions to tell you whether or not something is "learner-centered". For me, the meaning of "learner-centered" was summed up on the day that LaToya wrote her diver-program. As she sat transfixed, cheering her little stick-figure's comical swan-dive, I didn't need to check my district standards to know the value of her learning experience. It was simple: the bell rang, class was over, and LaToya wouldn't leave!

Once you've had a chance to see the power of FUNdaMENTAL at work in your classroom, it's stories like these - and not the wisdom of researchers –that you'll find yourself repeating. You'll tell them to colleagues, to parents and administrators…to anyone who will listen. And once you do, it's not likely you'll have much else to explain. The proof is in the programming!

# APPENDIX B
## FUNdaMENTAL Programmer's Tool Kit



This appendix contains some basic tools you and your students will come back to again and again as you explore the world of FUNdaMENTAL programming.

In section B1 you'll find a list of all FUNdaMENTAL commands, each presented with its function and data requirements. This is the same information you can see for each command in the Task window, but it's useful to see all the commands explained together so you can make connections and comparisons as you learn the FUNdaMENTAL language.

In section B2 you'll find a glossary of special terms used in the manual, including those applying to programming in general and others that are specific to the FUNdaMENTAL environment.

In section B3 you'll find a information on all the FUNdaMENTAL interface features as well as some quick tips on program editing and debugging. *Please note* that the same information is available online. Simply click on any feature in the interface and press the F1 key on the keyboard.

# ALL-COMMANDS LIST

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| ADD NUMBER | Adds the specified number to the AC | NUMBER | <number or box> | ADD NUMBER 37 |
| AND BOOLEAN | Logically "ANDs" the specified boolean with the AC. | BOOLEAN | <boolean or box> | AND BOOLEAN TRUE |
| APPEND STRING | Appends the specified string to the AC. | STRING | <string or box> | APPEND STRING " is funny - looking." |
| BRING-FRONT OBJECT | Brings the object in the AC to the front of the playground. | OBJECT | None | BRING-FRONT OBJECT |
| COMPARE BOOLEAN | Compares the specified boolean to the AC; sets the c-bit accordingly. | BOOLEAN | <boolean or box> | COMPARE BOOLEAN TRUE |
| COMPARE NUMBER | Compares the AC to the specified number; sets the c-bit accordingly. | NUMBER | <number or box> | COMPARE NUMBER 55 |
| COMPARE OBJECT | If the object in the AC is identical to the inputted box, or if it is a "member" of the inputted object type, sets the c-bit to 0. | OBJECT | <object type or box> | COMPARE OBJECT "Blue Fish" |
| COMPARE STRING | Compares the AC to the specified string; sets the c-bit accordingly. | STRING | <string or box> | COMPARE STRING "Hello, World" |
| CONSTRUCT GAGGLE | Allocates boxes for a new gaggle with the specified number of items and puts it in the AC. | Doesn't care | <number or box> | CONSTRUCT GAGGLE 101 |
| CONSTRUCT OBJECT | Constructs a new object using the specified object "blueprint" and puts it in the AC. | Doesn't care | <object type> | CONSTRUCT OBJECT "GreenTriangle" |
| DESTROY GAGGLE | Destroys the gaggle in the AC (reclaiming its memory); you can't use the gaggle anymore after this call. | GAGGLE | None | DESTROY GAGGLE |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| DESTROY OBJECT | Destroys the object in the AC (reclaiming its memory); you can't use the object anymore after this call. | OBJECT | None | DESTROY OBJECT |
| DISSECT STRING | Takes the letters of the AC which are between and including the specified positions and puts them in AC. | STRING | <number or box>, <number or box> | DISSECT STRING bLen, 5 |
| DIVIDE NUMBER | Divides the AC by the specified number. | NUMBER | <number or box> | DIVIDE NUMBER 2 |
| END SUB-TASK | Causes the Sub-Task to end; Must have at least one at the end of all Sub-Tasks. | Doesn't care | None | END SUB-TASK |
| EXECUTE PROCESS-TASK | Triggers the specified task; note that this creates a new process. | Doesn't care | <task name> | EXECUTE PROCESS-TASK BounceBall |
| EXECUTE SUB-TASK | Executes the specified Sub-Task; does not create a new process. | Doesn't care | <task name> | EXECUTE SUB-TASK StartGame |
| EXIT PROGRAM | Causes the program to end; must be the last line of the Program Task. | Doesn't care | None | EXIT PROGRAM |
| GET-BOTTOM OBJECT | Puts the AC's bottom y-coordinate in the AC. | OBJECT | None | GET-BOTTOM OBJECT |
| GET-HEIGHT CONVERSATION | Puts the Conversation Window's height in the AC. | Doesn't care | None | GET-HEIGHT CONVERSATION |
| GET-HEIGHT PLAYGROUND | Puts the Playground Window's height (drawing area) into the AC. | Doesn't care | None | GET-HEIGHT PLAYGROUND |
| GET-LEFT CONVERSATION | Puts the Conversation Window's left x-coordinate in the AC. | Doesn't care | None | GET-LEFT CONVERSATION |
| GET-LEFT OBJECT | Puts the AC's left x-coordinate in the AC. | OBJECT | None | GET-LEFT OBJECT |
| GET-LEFT PLAYGROUND | Puts the Playground Window's left x-coordinate in the AC. | Doesn't care | None | GET-LEFT PLAYGROUND |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| GET-LENGTH STRING | Counts the number of letters in the AC and puts that number in the AC. | STRING | None | GET-LENGTH STRING |
| GET-PICTURE OBJECT | Puts the name of the AC's current picture in the AC. | OBJECT | None | GET-PICTURE OBJECT |
| GET-RIGHT OBJECT | Puts the AC's right x-coordinate in the AC. | OBJECT | None | GET-RIGHT OBJECT |
| GET-SIZE GAGGLE | Counts the number of items in the gaggle in the Gaggle Register and puts that number in the AC. | Doesn't care | None | GET-SIZE GAGGLE |
| GET-TOP CONVERSATION | Puts the Conversation Window's bottom y-coordinate in the AC. | Doesn't care | None | GET-TOP CONVERSATION |
| GET-TOP OBJECT | Puts the AC's top y-coordinate in the AC. | OBJECT | None | GET-TOP OBJECT |
| GET-TOP PLAYGROUND | Puts the Playground Window's top y-coordinate in the AC. | Doesn't care | None | GET-TOP PLAYGROUND |
| GET-WIDTH CONVERSATION | Puts the Conversation Window's width in the AC. | Doesn't care | None | GET-WIDTH CONVERSATION |
| GET-WIDTH PLAYGROUND | Puts the Playground Window's width in the AC. | Doesn't care | None | GET-WIDTH PLAYGROUND |
| GET LOOP | Puts the value of the Loop Register in the AC. | Doesn't care | None | GET LOOP |
| GET TICKS | Counts the number of "ticks" (1/60 seconds) since midnight and puts it in the AC. | Doesn't care | None | GET TICKS |
| HIDE OBJECT | Makes the object in the AC invisible in the Playground Window. | OBJECT | None | HIDE OBJECT |
| IMPRINT OBJECT | Imprints the picture of the object in the AC to the background. | OBJECT | None | IMPRINT OBJECT |
| INSTALL BACKGROUND | Installs the specified picture as the background of the Playground Window. | Doesn't care | <picture name or box> | INSTALL BACKGROUND "MyPicture" |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| JUMP < | Jumps to the specified Marker if last compare was less-than (will jump when c-bit is -1). | Doesn't care | <Marker name> | JUMP < @MyMarker |
| JUMP <= | Jumps to the specified Marker if last compare was less-than or equal (will jump when c-bit is -1 or 0). | Doesn't care | <Marker name> | JUMP <= @MyMarker |
| JUMP <> | Jumps to the specified Marker if last compare was not equal (will jump when c-bit is non-zero). | Doesn't care | <Marker name> | JUMP <> @MyMarker |
| JUMP = | Jumps to the specified Marker if last compare was equal (will jump when c-bit is 0). | Doesn't care | <Marker name> | JUMP = @MyMarker |
| JUMP > | Jumps to the specified Marker if last compare was greater-than (will jump when c-bit is 1) | Doesn't care | <Marker name> | JUMP > @MyMarker |
| JUMP >= | Jumps to the specified Marker if last compare was greater-than or equal (will jump when c-bit is 1 or 0) | Doesn't care | <Marker name> | JUMP >= @MyMarker |
| JUMP ALWAYS | Jumps to the specified Marker (always). | Doesn't care | <Marker name> | JUMP ALWAYS @MyMarker |
| JUMP FALSE | Jumps to the specified Marker (if the AC contains "FALSE"). | BOOLEAN | <Marker name> | JUMP FALSE @MyMarker |
| JUMP LOOP | Subtracts 1 from LR and "Jumps" to the specified Marker (if LR is greater than 0) | Doesn't care | <Marker name> | JUMP LOOP @MyMarker |
| JUMP TRUE | Jumps to the specified Marker (if AC contains "TRUE") | BOOLEAN | <Marker name> | JUMP TRUE @MyMarker |
| LOAD-ITEM GAGGLE | Loads the specified item from the Gaggle in the Gaggle Register into the AC. | Doesn't care | <number or box> | LOAD-ITEM GAGGLE 10 |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| LOAD BOX | Puts the contents of the specified box into the AC. | Doesn't care | <box> | LOAD BOX MyBox |
| LOAD NUMBER | Puts any regular old number like 12 or 495889 into the AC. | Doesn't care | <number> | LOAD NUMBER 14 |
| LOAD SOUND | Puts the specified sound into the AC. | Doesn't care | <sound name> | LOAD SOUND "Ahhhhhhhhhh!" |
| LOAD STRING | Puts a "literal" string into the AC (any words in quotes is a "literal" string). | Doesn't care | <string> | LOAD STRING "Hello, World" |
| LOWERCASE STRING | Makes the string in the AC all lowercase letters. | STRING | None | LOWERCASE STRING |
| MORPH OBJECT | Changes the graphic of the object in the AC to the specified picture. | OBJECT | <picture name or box> | MORPH OBJECT "MyPicture" |
| MOVE OBJECT | Moves the object in the AC the specified distance in the x (first input) and y (second input) directions. | OBJECT | <number or box>, <number or box> | MOVE OBJECT -5, 0 |
| MULTIPLY NUMBER | Multiplies the AC by the specified number. | NUMBER | <number or box> | MULTIPLY NUMBER 88 |
| NOT BOOLEAN | Performs the NOT operation on the AC | BOOLEAN | None | NOT BOOLEAN |
| OR BOOLEAN | Performs an "OR" on the AC using the specified boolean value. | BOOLEAN | <boolean or box> | OR BOOLEAN TRUE |
| PLACE CONVERSATION | Moves the Conversation Window to the specified location on the screen. | Doesn't care | <number or box>, <number or box> | PLACE CONVERSATION 10, 20 |
| PLACE OBJECT | Moves the object in the AC to the specified location in the Playground Window. | OBJECT | <number or box>, <number or box> | PLACE OBJECT 10, 20 |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| PLACE PLAYGROUND | Moves the bottom-left corner of the Playground Window to the specified location on the screen. | Doesn't care | <number or box>, <number or box> | PLACE PLAYGROUND 10,20 |
| PLAY-N-WAIT SOUND | Plays the sound in the AC through the speaker and waits until it is done playing. | SOUND | None | PLAY-N-WAIT SOUND |
| PLAY SOUND | Plays the sound in the AC through the speaker; does not wait for the sound to finish. | SOUND | None | PLAY SOUND |
| PREPEND STRING | Prepends (puts in front) the specified string to the AC. | STRING | <string or box> | PREPEND STRING "Hello, my name is " |
| RANDOM NUMBER | Picks a random number between (and including) the two specified numbers and puts it in the AC. | Doesn't care | <number or box>, <number or box> | RANDOM NUMBER 1, 10 |
| READ-FILE NUMBER | Reads a number from the specifiled file. | Doesn't care | <file name or box> | READ-FILE NUMBER "MyFile.txt" |
| READ-FILE STRING | Reads a string from the specified file. | Doesn't care | <file name or box> | READ-FILE STRING "MyFile.txt" |
| READ-SCREEN NUMBER | Waits for the user to type in a number in the Conversation Window and puts it in the AC (when <return> is pressed). | Doesn't care | None | READ-SCREEN NUMBER |
| READ-SCREEN STRING | Waits for the user to press <return> in the Conversation Window and puts what was typed in the AC. | Doesn't care | None | READ-SCREEN STRING |
| RECORD SOUND | Displays a sound-recording panel, waits for user to record and save a sound, and puts the sound in the AC. | Doesn't care | None | RECORD SOUND |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| REMAINDER NUMBER | Divides the AC by the specified number and puts the remainder in the AC. | NUMBER | <number or box> | REMAINDER NUMBER 66 |
| RESIZE CONVERSATION | Resizes the Conversation Window to have the specified width (the first inputted number) and height (the second number). | Doesn't care | <number or box>, <number or box> | RESIZE CONVERSATION 88, 400 |
| RESIZE PLAYGROUND | Resizes the Playground Window to have the specified width (the first inputted number) and height (the second number). | Doesn't care | <number or box>, <number or box> | RESIZE PLAYGROUND 88,400 |
| REWIND FILE | Rewind the specified file to the beginning. | Doesn't care | <string or box> | REWIND FILE "myFile" |
| SEND-BACK OBJECT | Sends the object in the AC to the back of the playground. | OBJECT | None | SEND-BACK OBJECT |
| SET GAGGLE-REGISTER | Puts the contents of the specified box into the Gaggle Register. | Doesn't care | <box> | SET GAGGLE-REGISTER myPhoneNumbers |
| SET LOOP | Sets the LR (Loop Register) to the specified value. | Doesn't care | <number or box> | SET LOOP 25 |
| SHOW OBJECT | Makes the object in the AC visible on the Playground Window. | OBJECT | None | SHOW OBJECT |
| SLEEP MAIN | Puts the Program Task "to sleep"; can only be called from the Program Task. | Doesn't care | None | SLEEP MAIN |
| STORE-ITEM GAGGLE | Stores the contents of the AC into the specified box of the gaggle in the Gaggle Register. | Anything | <number or box> | STORE-ITEM GAGGLE 10 |
| STORE BOX | Stores the contents of the AC in the specified box. | Anything | <box> | STORE BOX myBox |

| Command Name | What It Does | Thinks AC Is | Data It Needs | Example Usage |
|---|---|---|---|---|
| SUBTRACT NUMBER | Subtracts a number from the AC. | NUMBER | <number or box> | SUBTRACT NUMBER 99 |
| TOUCHING OBJECT | Sets the c-bit to 0 if the object in the AC is "touching" the inputted object. Otherwise, sets the c-bit to non-zero. | OBJECT | <box> | TOUCHING OBJECT bullsEye |
| UPPERCASE STRING | Converts the string in the AC to all uppercase letters. | STRING | None | UPPERCASE STRING |
| WAKE MAIN | Wakes the main task (if it was asleep); can only be called from a non-main task. | Doesn't care | None | WAKE MAIN |
| WRITE-FILE NUMBER | Writes the number in the AC out to the specified file. | NUMBER | <file name or box> | WRITE-FILE NUMBER "MyFile.txt" |
| WRITE-FILE STRING | Writes the string in the AC out to the specified file. | STRING | <file name or box> | WRITE-FILE STRING "MyFile.txt" |
| WRITE-SCREEN NUMBER | Writes the number in the AC out to the Conversation Window. | NUMBER | None | WRITE-SCREEN NUMBER |
| WRITE-SCREEN STRING | Write the string in the AC out to the Conversation Window. | STRING | None | WRITE-SCREEN STRING |

# FUNdaMENTAL Quick-Reference Guide

## The FUNdaMENTAL Interface



## Editing Fm Programs

## Graphic Library

Graphic: Show the name of the currently selected graphic. You may rename the current graphic by double-clicking on its name and typing in another name.

Graphic List Box: Shows the name of all available graphics in the program.

Viewing Area: Shows the currently selected graphic.

New: Use this to create a new "empty" graphic. After creating a new graphic name you may link it to graphics which you paste from the Windows clipboard into the viewing area.

Delete: Click here to delete an unwanted graphic.

Import: Click here to import graphics from other places (e.g. the FUNdaMENTAL graphics archive.) See also Graphic Importer

## Welcome Window

Create New Program: Clicking on this sign will allow you to create a new FUNdaMENTAL program for code writing and editing.

Open Existing Program: Clicking on this sign will allow you to open a pre-written FUNdaMENTAL program for code editing.

Gallery: Clicking on the castle takes you to the gallerywhere you may play completed FUNdaMENTAL programs. You cannot edit or see the code of any programs in the gallery Instead you simply play and enjoy completed programs. (Windows version only)

Quit: Exits FUNdaMENTAL

## The Gallery

Picture Preview: Shows a graphic representing your program. See also The Toolbar and the Add To Gallery menu item

Current Programs List: Displays all the programs currently in your gallery

Play!: Play a program by finding it within the picture frame and then click once to select it. When selected, a black rectangle will outline the program. Then click the "Play" button. You may also play the program by double-clicking on it.

Remove: If you do not wish for a certain program to be in the Gallery, click once on it to highlight it in the Current Programs List, and then click the "Remove" button.

# The Toolbar

(See also Toolbar Buttons)

## Menu Items

### File

Close Program: Closes the current program and returns to Welcome Window
Save Program: Saves the current program
Close Window: Closes the current Task Window
Add To Gallery: Puts the current program into the Gallery
Make Plug-In File: Makes the current program into a special file that is viewable on Internet browsers that support plug-ins. (e.g. Netscape) See also Plug-Ins
Quit: Quit FUNdAMENTAL

### Edit

Undo: Undoes the last cut, copy, paste or delete.
Cut: Cuts the selected text into the FM clipboard. See also Cut n' Paste
Copy: Copies the selected text into the FM clipboard
Paste: Pastes the selected text from the FM clipboard
Delete: Deletes the selected lines

Insert Line: Inserts a line at the highlighted point in the Instruction List
Delete Line: Deletes the highlighted line.

### Languages

Switches the current language of FM. Currently English and Spanish are available.

### Task

Rename Task: Renames the current task. (If the Program Window is the topmost or ACTIVE window then the current task is the task highlighted in the Program Window. If Task Window is the ACTIVE window, then it is the current task.)
Syntax Check: Checks the syntax of the current task.
Print Task: Prints the current task.

### Program

Go: Starts the current program
Pause: Pauses the current program

Abort: Stops the current program.
Step: Go one instruction step. See also Debugging.
New Task: Creates an new empty task.
Add Task: Adds a prewritten sub-task to the current program. See also Task Importer
Delete Task: Removes a sub-task from the current program.

(Note these are in the Program menu because these are thing that affect the entire program and not just one single Task.)

### Window

Program Window: Shows the Program Window
Playground Window: Shows the Playground window .
Conversation Window: Shows the Conversation window
Debugger: Shows the Debugger window.
Object Designer: Shows the Object Designer
Sound Room: Shows the Sound Room
Graphic Library: Shows the Graphic Library
Clean Up All: Neatly arranges the open Task windows and also places the Program Window and Toolbar in a tidy order on the desktop.
[Current Windows] : Shows a list of all the currently open Task Windows. Selecting one will bring that Task Window to the top of the desktop

### Help

How to use FM: Shows this interface help on-line
Search: Starts a search for a topic or keyword in this FM help file.
FM Manual: Shows the on-line teachers manual for FM..
About KSI: Tells about the people who brought you FUNdaMENTAL!

## Toolbar Buttons

Copy: Copies the currently selected text. See also Cut n' Paste
Cut: Cuts the currently selected text to the FM clipboard.
Paste: Pastes code from the FM clipboard.
Play: Plays the current program.
Pause: Pauses the current program.
Stop: Stops the current program.
Step: Executes a single FM instruction. See also Debugging
Debug: Opens the Debugger window
Grid: Shows a grid on the Playground Window. This is useful for the placement of objects in the Playground.

## The File Menu

Close Program
Save Program
Close Window
Make Plug-In File
Add To Gallery
Quit

## Object Designer

Graphic: Click here to open a pop-up menu of all available graphics you can associate with an object. The selected graphic can be seen to the right of the Graphic pop-up menu. See also Graphic Library

Click Task: Click here to open a pop-up menu of all available sub-tasks to execute when the given object is clicked with the mouse.

Ctrl-Click Task: Click here to open a pop-up menu of all available sub-tasks to execute when the given object is ctrl-clicked with the mouse.

X-Pos: Specify the initial x position of a given object in the Playground Window. The default is zero. See Also Playground Window

Y-Pos: Specify the initial y position of a given object in the Playground Window. The default is zero. See Also Playground Window

Object: This text field shows the name of the currently selected object. Objects are selected by clicking on an object's name in the list below the Object text field. You may also rename objects here.

Object List Box: This is the area where all objects available to the current program are displayed. To select an object and see/change it's properties (e.g. name, graphic, click-task, etc) scroll to it in the list box and click on it.

New: Creates a new object. When pressed this will be given a default name Object_X where X is a number. We recommend you name all objects you create in a more descriptive manner.

Delete: When you have selected an object in the Object List Box, you may delete it by clicking on the Delete button.

Done: When you are finished designing your objects, click here to close the object designer.

(See Also Creating New Objects)

## Creating New Objects

To create a new object, you must first have at least one graphic available in your program's Graphic Library. Please see the Graphic Library if you are unsure about how to check this.

Creating that new object

1) Click ctrl-M to open the Object Designer. Alternatively you can go to the Window menu on the Toolbar and select "Object Designer" or .click on the Object Designer icon on the Program Window

2) Click the "New" button. A new object will be created named "Object_1".

3) If you'll notice, the graphic associated with the object will be the first one alphabetically in your Graphic Library collection. If you want to use that one great, go on to step #5.

4) Otherwise you can choose any picture you want... In the drop-down list labeled "Graphic" click on the down arrow and scroll until you see the name of the graphic you want. When you find it, click on the name and you will see a little preview to the right of what your graphic is going to look like.

5) For now don't worry about the "Click-Task" and "Ctrl-Click-Task" drop-down lists or the "x-pos" and "y-pos" boxes

6) Since "Object_1" is a poorly descriptive name, you should rename your new Object. Do this by clicking to the left of the 'O' in "Object_1". ( "Object_1" is in the text box labeled "Object:") Click and drag (i.e. keep that mouse button held down) until you have highlighted all of "Object_1" in the box.

7) Type a new descriptive name for the object.

8) Click the "Done" button.

# Graphic Importer

Use the graphic importer when you would like to import one or more graphics into your program at once.

Directory: Displays the current directory. You may navigate through the directory structure via this box.

Drive Selector: Below the "Directory" box is a pop-up menu where you can select among your computers hard disk, a floppy disk, or even a CD rom.

Picture Files: Display all the available graphics in the current folder of the Directory box. Click on a graphic file to select it or double click on it to add it to the "Graphics to Add" box.

Add: Click this button after you have selected a graphic in the "Picture Files" box to add to the "Graphics to Add" box.

Delete: You may remove a graphic from the "Graphics to Add" list using this button. Simply click on the offending entry to highlight it and then click "Delete"

## Importing graphics in 5 easy steps:

1. First navigate in the "Directory" box until you reach the folder in which your graphic file resides. FUNdaMENTAL only recognizes graphic files that end with a ".bmp" suffix. (Note you may also grab files off a floppy disk or CD-ROM using the drive box, which is directly below the "Directory:" box.)

2. When you reach a folder with graphic (*.bmp) files, then they will appear listed in the "Picture Files" box. Click once on any listed in the "Picture Files" box and you will see a preview of the graphic in the upper left hand corner of the Graphic Importer.

3. Next click the "Add" button to build a list of files you would like to add to your FUNdaMENTAL program. You may also double click on anything in the "Picture Files" box to add them. All added graphics will appear in the "Graphics to Add" box.

4. If you make a mistake and don't want to include a picture, click on the picture name in the "Graphics to Add" box and then click the delete button.

5. When you have finished building your list of graphics click the "Ok" button. If you decide you don't want to import anything just now, then click "Cancel".

# Directory Boxes

Frequently in FUNdaMENTAL, you will be asked to navigate through directories using a window similar to the picture above. This is a necessary skill so if you' feel a little shaky or don't know what we're talking about then read on.

These seemly complex windows are for navigating through the *directory structure* of your computer. A computer stores all it's files much like a typical file cabinet you would find in any office. Instead of randomly placing all its papers haphazardly in the file cabinet, people will usually arrange similar or related items in a folder. The computer provides for the same functionality.

The *directory structure* is arranged in a *hierarchy* of folders or *directories*. Unlike actual folders, an computer folder may contain other folders. Any given folder or "directory" in you computer's hierarchy may contain files, other folders or a combination of the two. Each time you double click on a directory (Directories are in the right hand side scroll box) any subfolders will be displayed underneath. Sub folders are beneath and indented a little to the right from its parent folder

The *root directory* is folder that contains all other folders. For your hard disk, the root directory is the one labeled "c:". The *root directory* can be thought of as the folder that contains all other folders and files. Some people like to imagine this *hierarchy* structure as a tree. The "c:" folder is the root the subfolders in "c:" can be thought of as branches from "c:" Further all the folders in c: may themselves contain subfolders.

The way you designate something in the directory structure is from the root up. Suppose we have a file called "huh". The file "huh" resides in a directory called "easy", which resides inside a folder called "is". In turn "is" resides in a folder called "This" And finally "This" is a subfolder of the "c:" We would designate the *path* to the file "huh" as "c:\This\is\easy\huh" Notice each directory is separated from its *parent directory* by the backslash('\') character.

Suppose we are looking for the file c:\fundam\manual\practice\demo1\demo1.fmp. With the tree structure in mind, you navigate through the *directory structure* as follows

1. Make sure the "Drives:" pop up box is on the drive labeled "c:" If it's not, then click on the down arrow of the "Drives" box with the mouse and all available drives will pop up. Choose the one the corresponds to your hard drive. In most cases this will be "c:"

2. First scroll to the top of the "Files" box. (Sometimes it will be called "Directory") Find the folder labeled "c:". Double click on this folder. You will now see all the sub directories of c: listed.

3. Scroll down until you find the sub folder “fundam”. When you find it, double click on it to reveal its contents. Repeat until you have reached the demo1 folder.

4. You may have noticed the box labeled “File Name:” now displays the files in “c:\fundam\manual\practice\demo1\” Depending on the type of window you are looking at, this box may either be to the right or left of the directory box. Move the mouse over the “demo1.fmp” entry and click on it. You have now instructed the computer to puts it’s attention on the file “c:\fundam\manual\practice\demo1\demo1.fmp.”

From here you can do many things, like Open a New Program, add a graphic, or even add a sound to your program.

## Task Window

This is the main area for editing your FUNdaMENTAL programs. The Task Window is divided into two main sections, the INSTRUCTION WIZARD and the CODING AREA. This will be primary focus for creating and editing all FM programs. In creating a new program or opening an existing one, you will see a programs "main" Task Window. Other sub-task windows may be viewed by using the PROGRAM WINDOW.

See also TOOLBAR.

## Instruction Wizard

Command Groups: As you are learning FM, you will find it useful to see the various commands sorted into related groups. For instance, a might want to see everything she can do with Objects. So you scroll down this pop-up list until you reach "Objects". The Instruction Wizard will filter out every command except the one's pertaining to Objects.

All Commands: This window will alphabetically display all commands, depending on the setting of the "User level/Commands Grouping" box. If you click on an entry, notice that it appears in the "Text Entry" box.

Instant Help: The upper left hand corner of the Task Window displays the Instruction Wizard's "Instant Help". It answers all the important questions about a FM command: What the Command is; What data the AC should contain; What additional data (or parameters) it requires; and shows appropriate usage for the command.

See also TOOLBAR

# Coding Area

Text Entry: This is where all new commands are entered into the "Instruction List". If you type directly into the "Text Entry" notice that the Instruction Wizard will show all possible commands that match what you have typed so far. So if you type an 'e', then the Instruction Wizard will just to the section of all commands that start with the letter 'e'. In this case, it will high-light the command "EXECUTE PROCESS-TASK" in the "All Commands" window and also put "EXECUTE PROCESS-TASK" in the "Text Entry" area.

Use: Click this button to use the current command displayed in the "Text Entry" area. In some cases it will bring up the DATA WIZARD

Comment/Instruction/Label: This set of radio buttons determine the type of code the user will enter. Only one button can be depressed at a time. See EDITING FM PROGRAMS

Instruction List: Here is where you may view the current task code of your FM program. All commands entered from the "Text Entry" box will appear here. You may also edit existing lines, cut and paste code lines, or set DEBUGGER break points from this section. See also EDITING FM PROGRAMS and DEBUGGING PROGRAMS

Boxes: This window contains all the boxes associated with the current Task Window. The icon next to the box name describes the SCOPE of the box. (A globe refers to global, a house to local scope.) To define a new box, click on the bottom of the list of boxes (or the first line if none are defined yet.) and an empty highlight should appear. Then simply type a new name. To change the SCOPE, click on the icon, switching it between a globe and a house. See also USING BOXES IN FM

Received Boxes: Sub-tasks may be written to use "Received Boxes" or parameters. THIS AREA WILL NOT APPEAR IN YOUR PROGRAM'S MAIN TASK. The icon next to a received box's name indicates the type of "received box" it is. (A one-way arrow denotes an "in" parameter. A two-way arrow denotes an "in/out" parameter.) To create a new received box, click on the bottom of the list of boxes (or the first line if none are defined yet.) and an empty highlight should appear. Then simply type a new name. To change the *RECEIVED TYPE*, click on the icon, switching it between a one-way arrow and a two-way arrow. See also USING BOXES IN FM

Key Task Button: Use this button to assign a key to a particular sub-task. THIS BUTTON WILL NOT APPEAR IN YOUR PROGRAM'S MAIN TASK. See also CREATING KEY TASKS

See also TOOLBAR. EDITING FM PROGRAMS

# EDITING FM PROGRAMS

When editing FM programs you may enter in three types of code.

- Comments
- Instructions
- Markers

## Adding/Editing Comments

Why comment in your code?

Although comments do nothing to the functionality of the program, comments are an integral part to good programming. Comments are lines of code in an FM program which aren't executed. Comments are denoted in green and surrounded by parenthesis. Good commenting will describe what your FM code is supposed to do. Not only does this help other people understand what you wrote when they look at your program, but it will also help you, serving as a reminder of what you wrote.

Consider the following snippet of FM code:

```
@top
    LOAD BOX counter
    ADD NUMBER 1
    STORE BOX counter
    COMPARE NUMBER 100
    JUMP < @dont_reset_counter
    LOAD NUMBER 0
    STORE BOX counter
@dont_reset_counter
    JUMP ALWAYS @top
```

It is rather hard to see what this is doing. However adding these three lines of comments will help to describe what the code does.

```
    (This code will simulate a counter that goes up to 100
    and then resets to zero. After reset, the counter starts to
    count upwards again.)
```

## To add a comment:

1. Make sure that the "comment" radio button is depressed. You should see that the "Text Entry" field and the highlighted area in "Instruction List" will have a pair of parenthesis showing "()".

2. Type in whatever comment is appropriate and then click the "use" button or press the <enter> key on your keyboard.

*Programmer's Hint: Often you would like to experiment by deleting a given command to see what effect it will have in a program. Instead of deleting the command line altogether, you can "comment" it out. To do this, highlight the line of code you wish to take out and click the "comment" radio button. Press the "use" button or <enter>.*

*Notice that instruction is now green and enclosed in parentheses. The next time the program runs, this line will not be executed. To change it back to an instruction, simply highlight the line again and click the "instruction": radio button.*

## Adding/Editing Instructions

Quite possibly, this is the most important skill to master when using FM. In some ways, editing FM programs is similar to using a spreadsheet. (See also CUTTING AND PASTING) In other ways, it's even easier.

To begin editing, make sure the "Instruction" button is depressed. Click inside the "Instruction List" Start by clicking where you would like to insert a new command.

Thanks to the INSTRUCTION WIZARD all you need in order to enter a new command is to type the first few letters. For instance, suppose you wanted to enter the command, LOAD NUMBER. All you would need to type is L-O-A-D-<space>-N and the Instruction Wizard will find the entire command LOAD NUMBER.

An added bonus of the "predictive" abilities of the Instruction Wizard, is that it will keep you from mis-typing the FM commands. If you were in a hurry and type L-O-D, the Instruction Wizard will give you beep to remind you that there is no FM command that starts with L-O-D.

Instead, it will show you all the closest commands starting with the letters L-O. Notice that as you type, the first command alphabetically (e.g. LOAD BOOLEAN) shows up in the "Text Entry" area.

Either continue typing the rest of the command or use the mouse to scroll in the "All Commands" box and click on the command you want to use. (You may also "up" or "down" arrows to select the command you want.) When the command shows up correctly in the "Text Entry", then hit <enter> on the keyboard or click the "use" button.

In some cases, a DATA WIZARD window will appear after you "use" a command. Fill in the request of the DATA WIZARD and then click OK.

Now your command should appear where the blue highlight was. Notice that the blue highlight is now directly below you're newly entered line.

Example:

1. Enter the FM command LOAD NUMBER 23.
2. Place the blue highlight by clicking where you would like to insert the line in the "Instruction List"
3. Type L~O~A~D [hit the space bar] N. This should make the Instruction Wizard select the command LOAD NUMBER.
4. Click the "use" button or the <enter> key on your keyboard.
5. In the Data Wizard that pops up, type in 23.
6. Click the OK button on the Data Wizard.
7. See Also USING BOXES IN FM. CUTTING AND PASTING, DATA WIZARD

# USING BOXES IN FM

In FUNdaMENTAL there are two classes of boxes: regular boxes and received boxes. Regular boxes are used for storage of data like numbers, objects, strings and such. Received boxes are used as parameters for sub~tasks.

In a sub~task's "Task Window" you will see area in the lower right for regular "Boxes" and "Received Boxes". (Note the main task has no "Received Boxes" area since the main task cannot take any parameters.)

You may add boxes two different ways.

## Using "STORE BOX"

When you type in a new command for "STORE BOX" a DATA WIZARD will pop up prompting for a box to store data into. You may either select a box already in the list or type the name of a new one.

If you type in the name of one that is not in the list, then another FM window will appear asking if you would like to define the box. Select yes. You'll notice that it will appear in the "Boxes" section of the current Task Window with a global icon.

### Editing within the "Boxes" area

You may also work directly within the "Boxes" area to add new boxes or edit existing boxes. Simply go to the bottom line of the list of boxes (If there are no boxes, then click on the first line.) and click in the middle of the line. You will see a blank highlight appear. This is a prompt for you to type in a new box name. After you finish, hit the <enter> key. Notice that your newly created box will be alphabetically sorted with the others.

As a beginner you should accustom yourself to using this area. In addition to adding new boxes, you can change the names of boxes or delete existing boxes, as well as define the scope of the box.

To delete a box: Click on the box name you would like to delete. It will be surrounded by a highlight. Hit the <del> or <backspace> key.

To change the name of a box: Highlight the box name you would like to change by clicking on it twice. You'll see a blinking cursor at the end of the box name. Use the backspace to delete and type in a new name.

To change the box's scope: Click on the icon next to the box's name. For boxes in the "Boxes" section, it will toggle between a globe (Global box) and a house (Local box). For boxes in the "Received Boxes" section it will toggle between a one-way and two-way arrow. See also SCOPE.

## Scope of Boxes

Within the two classes of regular and received boxes, there are two types of regular boxes and two types of received boxes. These types are defined by the scope of the box.

### Regular boxes (global vs local scope):

It may help to first clarify what the word scope means. Think of 'scope' as *the permission to use a box* given to a task. A task *uses* a box by storing data or loading data from a box.

A box is *global in scope* if *every* task in the program has permission to use it. It is designated by a globe icon in the "Boxes:" section of a Task Window. Notice that a global box will appear in all Task Windows of a given program. (To open multiple Task Windows see the PROGRAM WINDOW.) You can also think of a global box as a box that can be *seen* by all the tasks of a given program.

A box is *local in scope* if *only one* task in the program has permission to use it. It is designated by a house icon in the "Boxes:" section of a Task Window. Notice that a local box defined in one Task Window will not appear in any other Task Windows of a given program. (To

open multiple Task Windows see the PROGRAM WINDOW.) You can also think of a local box as a box that can be *seen* by only the task in which it is defined.

### Received Boxes (in parameters vs. in/out parameters)

If you want to pass in data to a sub-task, but don't want that data to change, then it must be an "in" parameter. An "in" parameter passed in to a sub-task cannot be changed by the sub-task. All types of data and boxes can be "in" parameters.

If you want to pass in data to a sub-task that needs to be altered, then you must pass it as an "in/out" parameter. Only boxes may be "in/out" parameters.

## Program Window

The Program Window is the Grand Central Station of FM. You will be able to access all Task Windows as well as the Sound Room, Graphic Library, and Object Designer.

New: Creates a new sub-task to be added into the program.

Add: Imports a premade sub-task into the program. See also Task Importer

Delete: Removes a sub-task from the program.

Task List: Lists all the available sub-tasks in the currently open program. To open the Task Window for a particular sub-task, double-click on the name.

Sound Room Icon: Click on this icon to open the Sound Room

Graphic Library Icon: Click on this icon to open the Graphic Library.

Object Designer Icon: Click on this icon to open the Object Designer

## Sound Room

Play: Plays the selected sound.

Stop: Stop playing the currently selected sound.

Record: If you have a microphone, you can record sounds through it by clicking on this graphic.

Sound: Displays the currently selected sound.

Sound List Box: Shows all available sounds in the program. To select a sound for playing, scroll around in this list until you find the sound you want and then click on it.

New: This button is for creating names for new blank sounds. New sounds may be linked to these newly created sounds by pasting from the clipboard or clicking the "Record" button.

Delete: Deletes an existing sound in the program.

Import: Imports a windows *.wav file from your computer's hard disk (e.g. the FUNdaMENTAL sounds archive). See also Sound Importer.

## Sound Importer

Use the sound importer when you would like to import one or more sounds into your program at once.

Play: Plays a selected sound.

Stop: Stops the current sound playing.

Directory: Displays the current directory. You may navigate through your the directory structure via this box.

Drive Selector: Below the "Directory" box is a pop up menu where you can select among your computers hard disk, a floppy disk, or even a CD rom.

Sound Files: Displays all the available sounds in the current folder of the Directory box. Click on a sound file to select it or double click on it to add it to the "Sounds to Add" box.

Add: Click this button after you have selected a sound in the "Sound Files" box to add to the "Sounds to Add" box.

Delete: You may remove a sound from the "Sounds to Add" list using this button. Simply click on the offending entry to hi-lite itand then hit "Delete"

### Importing sounds in 5 easy steps:

1. First navigate in the "Directory" box until you reach the folder in which your sound file resides. FUNdaMENTAL only recognizes sound files that end with a ".wav" suffix. (Note you may also grab files off a floppy disk or CD-ROM using the drive box, which is directly below the "Directory:" box.)

2. When you reach a folder with sound (*.wav) files, then they will appear listed in the "Sound Files" box. Click once on any name listed in the "Sound Files" box. You may preview the sound by clicking on the "play" button.

3. Next click the "Add" button to build a list of files you would like to add to your FUNdaMENTAL program. You may also double click on anything in the "Sound Files" box to add them. All added sounds will appear in the "Sounds to Add" box.

4. If you make a mistake and don't want to include a sound, click on the sound name in the "Sounds to Add" box and then click the delete button.

5. When you have finished building your list of sounds click the "Ok" button. If you decide you don't want to import anything just now, then click "Cancel".

## Task Importer

Use the task importer when you would like to import one or more tasks into your program at once. Often programmers will find themselves doing the same thing over and over, like sorting a gaggle of numbers. In such cases, it is often possible to reuse the same sub-task in various different programs.

Directory: Displays the current directory. You may navigate through the directory structure via this box.

Drive Selector: Below the "Directory" box is a pop up menu where you can select among your computer's hard disk, a floppy disk, or even a CD rom.

Task Files: Displays all the available tasks in the current folder of the Directory box. Click on a task file name to select it or double click on it to add it to the "Tasks to Add" box.

Add: Click this button after you have selected a task in the "Task Files" box. This will include the file in the list of files in the "Tasks to Add" box.

Delete: You may remove a task from the "Tasks to Add" list using this button. Simply click on the offending entry to highlight it and then click "Delete"

### Importing tasks in 5 easy steps:

1. First navigate in the "Directory" box until you reach the folder in which your task file resides. FUNdaMENTAL only recognizes task files that end with a ".fmt" suffix. (Note you may also grab files off a floppy disk or CD-ROM using the drive box, which is directly below the "Directory:" box.)

2. When you reach a folder with task (*.fmt) files, they will appear listed in the "Task Files" box. Click once on any name listed in the "Task Files" box.

3. Next click the "Add" button to include the selected file in the list of files you would like to add to your FUNdaMENTAL program. You may also double click on anything in the "Task Files" box to add them. All added tasks will appear in the "Tasks to Add" box.

4. If you make a mistake and don't want to include a task, click on the task name in the "Tasks to Add" box and then click the delete button.

5. When you have finished building your list of tasks click the "Ok" button. If you decide you don't want to import anything just now, then click "Cancel".

## Data Wizard

The Data Wizard is a tool that helps you to input the additional data required by some commands. For most commands the Data Wizard's queries are simple. There four types of Data Wizards.

- Direct Data (e.g. number, strings, BOOLEANS, sound, pictures, objects etc..)

{bml dwloadn.bmp} In these types of Data Wizards you type in a direct value like 23 for a number or "Hello world\" for a string. For sounds, pictures, and objects, respectively, the Data Wizard will allow to you to choose from all those available (via the programmer's importing or creating them). See also Sound Room, Graphic Library, and Object Designer.

- Data from Boxes: (LOAD BOX, STORE BOX)

{bml dwfishbx.bmp} For these you specify a box from which to get the data from. Boxes may be regular or received. You may also define boxes using the STORE BOX command. See also Using Boxes in FM

- Data for operations (ADD NUMBER, MOVE OBJECT etc)

Data for these commands can be either Direct Data or Data from Boxes.

- Data for parameters

The most complex Data Wizard is one where you need to add information for parameters. First you need to select the sub-task you wish add parameter info for in the upper half of the window. Then in the lower half of the Data Wizard you may either select a box or enter Direct Data by clicking in the column next to the parameter for which you wish to provide data.

## Cut n' Paste

Frequently it is useful to cut, copy, and paste text from one Task Window to another. FUNdaMENTAL provides for cutting, copying

and pasting like many other editing programs. Cutting or copying text, sends it to a temporary clipboard that FM keeps for transferring text. (Note this is separate from the Windows clipboard.)

For example to copy a section of code, click-and-hold on the area where you would like to begin copying. Drag the mouse over the text you want to copy. (Even if the text area you want is greater than the displayed area on the Task Window, keep dragging. The window will automatically scroll.)

After you have highlighted everything you want blue, then hit 'ctrl-c' or choose "copy" from the "Edit" menu.

To paste, simple click on the Task Window's Instruction List in the place you want to insert text and hit 'ctrl-v' or select "Paste" from the "Edit" menu.

Use the same procedure to Cut and Paste line.

You may likewise delete things altogether (without sending them to the FM clipboard.) by hilighting a section and pressing <backspace> or <delete>

## Creating Key Tasks

Every sub-task (not the 'Main" task) may be associated with a key. That is, when the associated key is pressed, the sub-task is executed. To associate a key with a task, simply click on the button bearing a picture a picture of hand pushing a key that to the right of the "Use" button. It is located in the upper right portion of the Coding Area in any Sub-Task window.

The above dialog will appear prompting you for a key to associate with the task. Type the key and then push OK. Click cancel if you don't wish to associate a key with a task.

## The Debugger

See also Hints on Debugging

AC: Shows the current value of the AC.

Gaggle: Shows the current gaggle in the gaggle register.

Loop: Shows the value of the current loop register.

Glasses: Shows the current type of data in the AC.

C-bit: Shows the current state of the C-bit.

Local Boxes: Displays all local boxes. You may view the type and value of all boxes in the scope of the current task.

Global Boxes: Displays all global boxes. You may view the type and value of all boxes regardless of which task is in the current scope.

Expert: Allows for more advanced debugging options

Processes: Allows you to change the scope of the debugger to any available processes running.

Tasks: Allows you to change the scope of the debugger to any available processes running.

## Debugging Programs

Not even the best of us get our programs right the first time. Especially when the programs get large, keeping things straight can be a difficult task. Fortunately, FM is equipped with a debugger. A debugger is an instrument that slows down the execution of a program so you can watch the FM machine execute things step by step.

The main reason this works is that the FM machine does not do anything it's not told to do. Bugs are what occur when a person wants to make their program do something, but they wrote an incorrect command or set of commands to do this. Consider the following.

```
LOAD NUMBER 45
STORE BOX count
LOAD BOX count
ADD NUMBER 23
EXECUTE SUB-TASK KaBlooey(count)
```

What the user wanted to do is pass in the value of count + 23 to the sub-task KaBlooey. However, she forgot to store the value in the AC back into the box "count".

## Debugging a first lesson:

Set up an FM program with a sub-task KaBlooey and the above code. Notice that there are clear circles next to each instruction. These are where you may set *break points*. A *break point* is a signal to the FM machine to stop running the program at this point. A *breakpoint* which is set will be filled in red.

As in the diagram above, set a breakpoint at the line LOAD BOX count. Then push the play button on the toolbar or ctrl-G to start the program running.

The program will pause in its execution at the line . Notice that "LOAD BOX count" is the next instruction that FM will execute. It is marked by a blue arrow.

Experiment a little and look at the values in the AC, the data-type and the other parts of the debugger.

Now click the step button on the toolbar. Notice that the value of the AC has changed. It will indicate the AC has 45 in it.

Step again using the step button on the toolbar. The AC will indicate a value of 23+45 or 68.

Notice the value of count is still 45. This is what will get passed into the sub-task "kablooey". How do you correct this code?

Strategies: A buggy program is a lot like a polluted river . Pollution comes from a definite source, but it is often difficult to tell exactly what that source is. Unfortunately, even a small source of pollution somewhere far upstream can cause all kinds of havoc for the rest of the river. From where you happen to be looking, all you see is polluted water. So how do you find the source of the pollution?

Often the best way to start is to make a guess at where the source of the pollution is. This may involve actually backtracking to a point where you know the FM code is "clean" and working properly. Set a breakpoint in that location and then step through carefully looking for the moment the code becomes polluted.

Again, comments are a must for good debugging. A bug usually occurs when a person has an idea of what they want their program to do, but makes a mistake in implementing their idea in code. Having good comments, therefore, makes it easier for a person debugging to know what the code *should* do. Then they can step through carefully and compare what the code is *actually* doing.

Don't be afraid to tinker: The big thing to remember is the *FM machine only does what you have programmed it to do.* You will never break FM by typing an incorrect command. Program crashes don't occur because FM randomly decides to be ornery and not follow your code. Rather they happen because of some oversight the programmer made. This means YOU, the programmer are in complete control of fixing an ANY bugs you encounter in your FM program.

# Markers

Markers are "jump" points in FM code. They are used in conjunction with the JUMP commands.

To place a marker, make sure the radio button for marker is depressed. Then you'll notice that that the Text Entry box in the Coding Area has an '@' ("at" sign). Then just type in an appropriate name of the label you want.

Note: Labels may only be one word. Still you can be descriptive by separating things with underscores "_". An example is @turn_right. It's still technically one word because it contains no spaces between characters, but you can see it's meaning more clearly than "@turnright"

Labels must be defined before you use a command JUMP <some jump condition> @<some label>. This is because the Data Wizard will ask you to jump to an *existing* label if you type in a JUMP command.

# GLOSSARY

Accumulator (AC): The portion of the Central Processing Unit responsible for holding the current data. This is where the computer "looks" for the data to be processed when it encounters a command like ADD NUMBER or MOVE OBJECT.

ADD: (See ADD NUMBER in All-Commands List)

APPEND: To add string data to the end of a string in the AC. (See All-Commands List: APPEND STRING.)

Array: A gaggle with items all of the same type.

Binary number: A long string of 1's and 0's that the computer understands and to which we can attach more easily understood information like commands and data.

Boolean: A kind of data that always has a value of either "true" or "false," and that can be manipulated according to the rules of formal logic

Box: A chunk of computer memory allocated to hold either a copy of a data value (in the case of numbers, strings, sounds, and booleans), or a copy of a pointer to a data value (in the case of objects and gaggles). (See also Global box; Local box; Memory; Parameters; Passed-in; Received boxes; Variable)

Bug: An error in the program code which causes a program to run differently than intended, or stop running altogether ("crash").

Bus: The communication line between the CPU and all the external components such as the keyboard, mouse, monitor, and audio speakers..

Central Processing Unit (CPU): The part of the inside of the computer that processes all the instructions in a program, containing the AC, c-bit, gaggle register, loop register, and scheduler.

Click-task: A sub-task linked to a particular object design, and which contains a set of instructions to be executed only when the user clicks on any instance of an object made from that design.

COMPARE: (See All-Commands List: COMPARE NUMBER; COMPARE STRING; COMPARE OBJECT)

Control-click-task: A click-task that will only be executed if the Control key on the keyboard is pressed at the same time the mouse is clicked.

Code: The collective commands, instructions, markers and comments that make up any given program.

Comment: A text message placed in the code by the programmer to prompt, clarify, or explain surrounding code for the programmer's benefit and the benefit of anyone

else's who might be trying to read or work with the program. (Comments don't communicate with the computer at all and take no part in the execution of programs in which they appear.)

Command: A two- or three-word phrase containing a verb followed by a data type (as in LOAD STRING), computer component (as in SET GAGGLE-REGISTER), or C-bit reading (as in JUMP =). The basic building blocks of the computer language, commands can stand alone or be combined with other specifications to create the instructions that make up program code.

Compare bit (C-bit): The part of the CPU that registers the results of comparisons in terms of "equals" (=), "greater than" (>) or "less than" (<).

Conversation window: The window that appears with the Playground window while a program is running, and that is displays text and receives typed input from the user.

Coordinate system: The programmer's coordinate system is an "upside down" version of the standard, Cartesian coordinate system, with the point (0,0) in the *upper* left-hand corner.

Crash: The sudden, unexpected aborting of a program, usually due to an error in the code.

Data: The information types that are manipulated in a program; FUNdaMENTAL can manipulate six types of data: numbers, strings, objects, sounds, gaggles, and booleans.

Data Wizard: The FUNdaMENTAL feature that prompts you to fill in the necessary specifications for commands like LOAD STRING or PLAY SOUND.

Data Wizard dialog: The little box that appears superimposed on the Task window when the Data Wizard is activated during the writing of a FUNdaMENTAL program. This is where you can type in or select the specific information or data required to make your instruction complete.

Debugger: The FUNdaMENTAL feature (accessible by clicking the Stop Sign radio button to the left of each instruction in your tasks, clicking the Play button on the Toolbar, and then clicking the Debugger button on the Toolbar) that displays information about your program's inner workings, including the current contents of the AC and the listed contents of items in any gaggles you may be using.

Decomposition: The process of analyzing a problem "from the top down" and breaking it into smaller, more manageable components.

Define: To place a box in the list of recognized boxes for a particular program.

DISSECT: To remove the outer characters from a text string and retain those remaining in the center. (See All-Commands List: DISSECT STRING.)

DIVIDE: (See All-Commands List: DIVIDE NUMBER.)

END: See All-Commands List: END SUB-TASK.)

Execute/EXECUTE: The term for the computer's action in carrying out instructions, tasks, and whole programs. (See All-Commands List: EXECUTE SUB-TASK; EXECUTE PROCESS-TASK.)

EXIT: To terminate and "leave" a particular application or program. (See All-Commands List: EXIT PROGRAM.)

File/FILE: Any discrete package of information in any form, such as text (".txt"), graphics (".bmp"), or FM program (".fmp") that is stored on your computer's hard drive or on floppy disk. In the FM language "file" refers specifically to text files stored in the same folder with a program which that folder can "write out to" or "read in from". (See All-Commands List: WRITE-FILE... and READ-FILE...)

Flag: A piece of data such as a number or boolean that is combined with an informative box name (such as "hasblueeyes" or "practicewords") in order to allow the computer to deal with otherwise unrecognizable information within the context of a program.

Gaggle: A unit of memory made up of a specified number of sub-units, or "items" (basically a bunch of boxes strung together). (See also Array, Item, Structure.)

Gaggle Register: The part of the CPU that holds a pointer to a stored gaggle while the AC deals with the individual items of the gaggle.

GET: To get a numerical reading, and place the resulting number in the AC. (See All-Commands List: GET-LOOP; GET TICKS; GET-BOTTOM/TOP/LEFT/RIGHT PLAYGROUND; GET-BOTTOM/TOP/LEFT/RIGHT OBJECT.)

Global box: A box that is accessible to all the sub-tasks in a given program.

Graphic: An image that can be used as the visible component of an object or as a program background.

Graphic Importer: The FUNdaMENTAL interface feature that allows you to import graphics from other FM programs or from the CD-ROM. (Get there by clicking the Import button in the Graphics Library.)

Graphic Library: The FUNdaMENTAL interface feature where the graphics for a given program are defined with descriptive names and stored.

Hard drive: The main "static" memory device inside your computer which stores and retrieves permanent information, and which allows information to remain intact even when the computer is turned off.

Heap: A portion of the computer's memory that is separate from the box portion, and where objects and gaggles are stored and accessed through pointers. Since space in the Heap is only reserved *once the program is running* (unlike the STORE BOX command, which reserves space while the program is being written), the Heap is sometimes also called "dynamic memory."

Import: To bring existing graphical or sound files into a given program from an outside source, such as another FM program, a floppy disk, or a CD-ROM.)

Input : Basically any information the user gives to the computer, whether it's through typing on the keyboard or moving and clicking the mouse. Input can be an FM instruction or comment, or a click of the mouse which activates a click-task. (Other inputs include sounds recorded through a microphone, scanned-in pictures, etc.)

Instance: (See Object Instance)

Instruction: A line of program code comprised of a single command or a combination of a command and another input such as a string, number, or box name.

Instruction list: The portion of the Task window that contains the program code.

Instruction Wizard: The Task window feature that automatically searches the list of FUNdaMENTAL commands to find a match when the programmer begins typing in the text-entry field. It also displays the function of a selected command in the upper right-hand corner of the Task window.

Item: A sub-unit of storage space in a gaggle.

JUMP: The computer's action of breaking the regular, "line-by-line" execution of a program in order to "jump" to another designated place in the code. (See All-Commands List: JUMP LOOP/ALWAYS/=/<=/>=/</>/<>.)

Key-task: A sub-task that is only executed when a designated key on the keyboard is pressed.

LOAD: To put into the AC a particular piece of data or the data contents of a specified box. (See All-Commands List: LOAD BOOLEAN, LOAD BOX, LOAD NUMBER, LOAD SOUND, LOAD STRING.)

Local box: A box that is only accessible from within one sub-task.

Loop/LOOP: The repetition of a specified sequence of code; also short for Loop Register. (See All-Commands List: SET LOOP.)

Loop Register: The part of the CPU that keeps track of the number of repetitions left to be executed when the computer is "looping" through a specified sequence of code.

Main task: The main list of instructions from which the chain of all other tasks in the program originates. Every program has one.

Marker: The code type identified with an "@", which marks the place above any instructions to which the computer needs to "jump" in executing any JUMP command.

Memory: A physical piece of computer hardware. (often called RAM, or Random Access Memory.). It is the temporary storage device that holds information as long as the program is running. Memory just holds a bunch of 1s and 0s that are interpreted as different data such as numbers, string, and objects, etc.

Modularity: The quality of a sub-task that makes it self-contained for the purposes of readability and easier "debugging," or troubleshooting.

Morph: To change the appearance of an object. (See All-Commands List: MORPH OBJECT.)

MULTIPLY: (See All-Commands List: MULTIPLY NUMBER.)

Number: A FUNdaMENTAL data type; integer.

Object: A FUNdaMENTAL data type; a short form of "Object Instance" or "Object Design."

Object Design: A "blueprint" residing in the Object Designer and containing the specifications for a particular object-type, including name, initial graphic, initial Playground location, and object click-task and/or Control-click-task.

Object Designer: The FUNdaMENTAL interface feature that allows you to create and store object designs for a given program.

Object Instance: An instance or edition of an object type constructed within a program from a given object design.

Output: Any information, in any form (be it picture, text, sound, etc.), which the computer puts out as a result of what a human put in.

Parameter: The specific data sent into an empty, or "received," box in a sub-task.

Passed-in: A transient parameter quality that is specified in one task and put to work as a "received" variable in another sub-task.

Pixel: The smallest graphical element on the monitor. A monitor is made up of a bunch of tiny CRTs (cathode-ray tubes) which project points onto the screen. These points are pixels.

Place: To specify the placement of an object on the Playground grid. (See All-Commands List: PLACE OBJECT.)

Playground grid: The coordinate grid, visible by clicking the Grid button on the Toolbar, providing the system by which the computer processes instructions for the placement and movement of objects in the Playground. (Please note that the programming grid is the inverse of the standard, mathematical Cartesian coordinate system, with the point (0,0) in the *upper* rather than in the lower left-hand corner.)

Playground window: The FUNdaMENTAL interface feature that appears when you click the Play button on the Toolbar and that displays all the graphical aspects of a given FUNdaMENTAL program (the objects, background graphics, etc.)

PREPEND: To add string data to the beginning of a string in the AC. (See All-Commands List: PREPEND STRING.)

Pointer: The reference device used by the computer to access information about an object or a gaggle which (unlike strings, numbers, sounds, etc.) cannot be actually copied back and forth between the AC and memory, but rather remain stored in the Heap while their host program is running.

Process: A sub-task that runs independently of other sub-tasks, allowing you to have several different tasks running simultaneously. FUNdaMENTAL simulates this by running one process for a while and then switching quickly to another process. (See also Scheduler)

Process-task: A sub-task that is executed as a separate process during the running of the program and, therefore, appears to be running simultaneously with the other process or processes in the program.

Program (n.): A group of instructions given in a language the computer understands, and arranged according to particular rules so the computer can create the desired output.

Programming: The art and science of giving instruction to the computer in order to achieve human goals.

Program window: The FUNdaMENTAL interface feature (accessed through the Window menu on the Toolbar) that allows you to define new sub-tasks and access preexisting ones.

READ: (See All-Commands List: READ-FILE STRING/NUMBER, READ-SCREEN STRING/NUMBER.)

Read In: To bring information or data into the CPU.

Readability: The quality of any given passage of program code that makes it easy for a person to understand and work with it.

Reusability: The quality of a sub-task that allows it to be used in a variety of different programs with little or no alteration.

Received box: A box in a sub-task that receives a parameter from an EXECUTE SUB-TASK instruction in another task.

REMAINDER: (See All-Commands List: REMAINDER NUMBER.)

Run: The computer's action in executing all the instructions of a given program, *or* a person's action in causing the computer to execute a program by clicking the Play button on the Toolbar.

Scheduler: The part of the CPU responsible for making sure that all current processes in a given program get equal attention according to program specifications.

SET: To specify the contents of a register in the CPU. (See All-Commands List: SET LOOP; SET GAGGLE REGISTER.)

SHOW: To make an object visible in the Playground window. (See All-Commands List: SHOW OBJECT.)

SLEEP: To suspend the execution of the instructions in a task. (See All-Commands List: SLEEP MAIN.)

Sound: A FUNdaMENTAL data type.

Sound Importer: The FUNdaMENTAL interface feature (accessed by clicking the Import button in the Sound Room) which allows you to import preexisting sound files from other FM programs, or from the CD ROM.

Sound Room: The FUNdaMENTAL interface feature (accessed by selecting Sound Room from the Window menu on the Toolbar) that allows you to create, import and store sound data for any given FM program.

STORE: To allocate, or reserve, memory space. (See All-Commands List: STORE BOX.)

String: A FUNdaMENTAL data type; any "string" of keyboard characters placed between quotation marks.

Style: The overall quality of a program, defined by the degree of its readability and efficiency and by the elegance of its logic.

Structure: A gaggle with items of different types.

Sub-task: Any task other than the main task that forms part of a complete program.

SUBTRACT: (See All-Commands List: SUBTRACT NUMBER.)

Syntax check: The FUNdaMENTAL feature (activated when the Play button on the Toolbar is clicked) that does a quick check of the program before it is run in order to verify that all commands, boxes, markers, and data have been correctly defined during the designing of the program.

Task: A list of instructions constituting all or part of a computer program.

Task window: The FUNdaMENTAL interface feature that allows you to write instructions, markers, and comments; define boxes; and see an explanation of the function of each command in the FUNdaMENTAL language.

Ticks: A unit of time equaling 1/60 of a second. (The command GET TICKS puts into the AC the current number of ticks elapsed since midnight.)

Toolbar: The FUNdaMENTAL "control panel" containing the primary buttons and all of the menus for accessing other functions and features of FM.

Use button: The button found to the right of the text-entry field in the Task window which, if clicked after selecting a command, will either enter that command in the Instruction List, or bring up a Data Wizard dialog eliciting further input required for the use of the command. (Note: The same effect is achieved by double-clicking on a command where it appears in the list of all commands above the text-entry field in the Task window.)

Variable: A predefined placeholder for a particular type of data, which may hold a different value with any given execution of the instruction or program that contains it. In FUNdaMENTAL, boxes provide the means for creating variables.

WAKE: To reverse the suspending action of the SLEEP command and cause the computer to resume execution of the instructions in the main task. (See All-Commands List: WAKE MAIN.)

WRITE: To copy a specified text string or number either into the Conversation window or into a text file associated with a FUNdaMENTAL program. (See All-Commands List: WRITE-SCREEN STRING/NUMBER; WRITE-FILE STRING/NUMBER.)

# INDEX

## A



## B



## C

## D



## F

## G



## H



## K



## L

## M



## N



## O



## P



## S

## T



## V



## W