4404

ARTIFICIAL INTELLIGENCE SYSTEM

INTRODUCTION TO THE SMALLTALK-80 SYSTEM

*Please Check at the
Rear of this Manual
for NOTES and
CHANGE INFORMATION*

Tektronix®
COMMITTED TO EXCELLENCE

MANUAL REVISION STATUS

PRODUCT: 4404 Artificial Intelligence System Smalltalk-80 System

This manual supports the following versions of this product: Version T2.1.2

REV DATE	DESCRIPTION
DEC 1984	Original Issue
AUG 1985	Addition of NOTES Section

CONTENTS

ILLUSTRATIONS

SECTION 1

INTRODUCTION

Welcome to the world of the Smalltalk-80† system. Your 4404 Artificial Intelligence System (called the 4404 in this manual) supports the Smalltalk-80 System Version 2 release of the Smalltalk-80 language and environment. With the 4404 and the Smalltalk-80 language and its environment, you have all you need to explore Smalltalk programming and develop Smalltalk applications code. The 4404 can also serve as the "delivery vehicle" for the applications developed on it.

This manual has two primary goals: it instructs you in the invocation of the Smalltalk-80 system on the 4404, no matter what your background in the Smalltalk-80 system is, and it documents the implementation and programming aspects of the Smalltalk-80 system on the 4404. If you have little or no experience with the Smalltalk-80 system, then you should probably read through the entire manual. Section 2 gives a brief summary of the main features of the Smalltalk-80 system for those who are not acquainted with it and Section 3 is a Smalltalk tutorial on the 4404. If you are an experienced Smalltalk programmer, you can turn immediately to Section 4, Programming in the Smalltalk-80 System. In Section 4, you are given just what you need to get quickly into the Smalltalk-80 system on the 4404.

Regardless of whether you are a novice or an expert in Smalltalk programming, you need to how to operate the 4404 first. You should also know something about the 4404 operating system with which the Smalltalk-80 system communicates. You can achieve this understanding by reading through the 4404 Artificial Intelligence System User's Manual. The presentation of information in this manual assumes that you have read through the 4404 AIS User's Manual.

About This Manual

Here is a summary of what you will find in this manual:

- Section 1. This tells you what is in this manual and where to find it. It tells you about the Addison-Wesley books on the Smalltalk-80 system and describes the 4404 documentation.

---

† Smalltalk-80 is a Trademark of Xerox Corporation.

- Section 2. This is a brief introduction for those who are not acquainted with the Smalltalk-80 system. Section 2 presents the main features of the programming environment, language, and user interface.

- Section 3. This is a tutorial for the Smalltalk-80 system on the 4404. You learn how to bring up the Smalltalk-80 system from a powered-off 4404. You learn how the mouse interacts with the windowing user interface. You learn how to open up a workspace window, use the editor to type in some simple Smalltalk code, and compile and execute the code. You also learn how to find information about the over 200 Smalltalk classes and their messages.

- Section 4. Here you learn how to bring up the Smalltalk-80 system quickly. You are provided with hints and tips about programming. You learn the basic ideas behind Model-View-Controller, an important part of the Smalltalk-80 system that you need to understand to design and write sophisticated Smalltalk applications.

- Section 5. This section contains the implementation aspects of the 4404 version of the Smalltalk-80 system. How the Smalltalk-80 system and the 4404 operating system interact is described. Also, Smalltalk-80 virtual image enhancements are presented. And finally, this section discusses the few differences between the Smalltalk-80 System Version 2 (as it is described in the Addison-Wesley books) and the 4404 version.

- Appendices. In Appendix A, you find a list of Smalltalk classes. In Appendix B, there are tables describing Smalltalk internal character codes.

The 4404 Artificial Intelligence System Documentation

In addition to this manual, the following documentation comes with the standard 4404:

- 4404 Artificial Intelligence System User's Manual. This manual contains what you need to know to operate the 4404 controls. It also has an introduction to the 4404 operating system for the first-time user.

- 4404 Artificial Intelligence System Reference Manual. Here you find detailed information about all the operating system commands, system calls and utilities. The 4404's hardware is discussed here from the programmer's point of view. And finally, you find documentation for the 68000 assembler and linker, the C language compiler, the text editor EDIT, and the terminal emulator.

This manual assumes that you have gone through the 4404 AIS User's Manual. The user's manual gives a good introduction to the 4404 and the operating system that the Smalltalk-80 system is built on top of. Take the time to become familiar with the 4404 controls and the operating system. This requires only about an hour or so, especially if you are familiar with the UNIX† operating system or a UNIX -like operating system.

The other software options have their own documentation, which you receive when you order these options.

---

† UNIX is a Trademark of Bell Laboratories.

The Smalltalk-80 System Reference Books

Since the purposes of this manual are to document the Smalltalk-80 system on the 4404 and to give a brief, introductory-level look at the Smalltalk-80 system, two textbooks published by Addison-Wesley are highly recommended as supplementary material. These books are:

● Goldberg, Adele. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, 1984. (Called in this manual the "Goldberg book", but known among Smalltalk programmers as the "orange book".)

● Goldberg, Adele and David Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983. (Called in this manual the "Goldberg and Robson book," but known among Smalltalk programmers as the "blue book".)

The Goldberg book is an extensive introduction to the Smalltalk-80 system. There are many tutorial exercises that describe how to use the text editor, how to make pictures, how to find out about classes and messages, how to program by modifying existing code, how to create new classes of objects, how to find and correct errors, how to use the external file system to save changes you have made to the Smalltalk-80 image, and so forth.

If your first acquaintance with the Smalltalk-80 system is on the 4404, you should read through Sections 2 and 3 in this manual and then proceed to the Goldberg book. Use the 4404 and the Goldberg book to learn the Smalltalk-80 system. Most tutorial exercises in the Goldberg book work on the 4404 exactly as they are described in the book. (The differences are noted in Section 5 of this manual.)

The Goldberg and Robson book is a formal explanation and description of the Smalltalk-80 language. This includes not only the syntax of the language but also the classes of objects that make up the Smalltalk-80 virtual image. This book also contains in part four a detailed discussion of the implementation of the virtual machine. The Smalltalk-80 system virtual machine is usually implemented in a particular hardware environment as a combination of hardware registers and logic, and assembly language or microcode instructions. (When you run the Smalltalk-80 system on the 4404, you first load the software part of the virtual machine — called the interpreter — and the virtual machine then loads the virtual image that you have specified in the command line.)

If you expect to develop any Smalltalk applications, you should study the Goldberg and Robson book. This is the standard textbook describing the Smalltalk-80 language.

SECTION 2

A SMALLTALK-80 SYSTEM OVERVIEW

This section provides a brief look at the Smalltalk-80 system from the point of view of programmers and others who do not know the Smalltalk-80 system. If you already know something about the Smalltalk-80 system, you can skip directly to the 4404 Smalltalk-80 system tutorial in Section 3. If you are a proficient Smalltalk programmer, you should skip directly to Section 4.

You should refer to the Addison-Wesley Smalltalk books (mentioned in Section 1 of this manual) for a comprehensive tutorial introduction and reference to the Smalltalk-80 system.

What is The Smalltalk-80 System?

The Smalltalk-80 system is an interactive programming environment.

It is interactive because it features fast response to the usual tasks of program development: creating, compiling, debugging, and running code. The system is designed so that, usually, the only use of the keyboard is to type actual text. Other actions are accomplished by operating the mouse buttons to make selections from menus. The system is also visually-oriented and advantageous feature which allows you to work unhampered by the necessity to remember commands.

The system is a well designed environment because you interact with the system through a uniform interface to many different functions. It is a programming environment because the system is dedicated to creating, compiling, debugging, and running Smalltalk code. This makes the Smalltalk-80 system similar to some LISP systems and to the FORTH programming system, among others.

The User Interface: Mouse, Windows, and Menus

One of the most striking (and distinctive) features of the Smalltalk-80 system is the user interface. It is visually-oriented. If you move the mouse around on the mouse pad, an arrowhead cursor moves around the screen in concert with the mouse movements. You can "point" at something, select it by pressing a mouse button, and then do something with the selected object by pressing another mouse button or typing at the keyboard.

You note immediately upon entering the Smalltalk-80 system that information is organized on the screen by being surrounded with rectangular boxes (called windows). Some windows contain Smalltalk code; some contain pictures; some contain text: notes, instructions, warnings, etc. By experimenting with the mouse, you observe that a number of different boxes with words appear on the screen. These are the Smalltalk pop-up menus.

The Smalltalk-80 system maintains within itself data that keeps track of which information is in what window, and, of course, it knows where the arrowhead cursor is on the screen and what mouse button has just been pressed or released. This data is constantly updated and is used to allow you to quickly select windows or the contents of windows and apply pop-up menu commands to them. Only the pop-up menus appropriate to the placement of the mouse appear. This helps focus your attention on the task at hand.

The Mouse

You use the mouse to direct the major activities of the Smalltalk-80 system. With the mouse, you select an object and then select from a pop-up menu what to do with the object. You can select windows, menu choices, and text within windows. The mouse has three buttons.

- The left button (the red button in the Addison-Wesley Smalltalk books) is used for selection. Its primary use is for selecting text in a window, moving the text position cursor in a window, or selecting an entire window (to be the active window).

- The middle button's (the yellow button in the Addison-Wesley Smalltalk books) primary use is to pop up a menu appropriate to the location of the mouse; it is context-sensitive. You select from the menu and the command is performed. The middle button menu commands apply to the contents of a currently selected window. For example, you can do things like cut, paste, and copy blocks of text in some windows.

- The right button (the blue button in the Addison-Wesley Smalltalk books) generally pops up a menu with commands that apply to the window itself and not to the contents of the window. You can do things like moving, framing, and closing windows.

Special-Purpose Windows

There are a number of special purpose windows to make your life easier in the Smalltalk-80 system. Among these are the System Browser, the System Transcript, and the System Workspace. There are other system-created windows, but these are the main ones.

- The System Browser window is your key to finding information in the system. Since the Smalltalk-80 virtual image consists of over 200 classes, you need to be able to find out about them quickly and easily. The System Browser window shows you not only the system classes but also all the messages that can be sent to them. Classes and messages are both categorized to narrow your search for information as much as possible. You also use the System Browser window for adding new classes and methods, and for modifying existing classes and methods. (See later in this section for a description of classes and methods.)

- The System Transcript window collects text messages from the system or from programs that you write. It allows you to monitor certain functions in the system. For example, when you recompile classes, messages about this action automatically appear in the System Transcript.

- The System Workspace window contains information about the many functions and Smalltalk-80 language "templates" appropriate to them. Templates are examples of Smalltalk code that you edit and compile to get information you are seeking. You find templates for how to open, read, and edit files, how to do system crash recovery, how to manage the system image file, how to interrogate the system, how to manage the changes you make to the system as you program, etc.

Pop-Up Menus

A workspace window is a window that you can use to develop new Smalltalk code. In this kind of window pressing the middle button of the mouse while the arrow cursor is within the borders of the window makes a pop-up menu appear. This menu has selections such as the following:

- copy — This puts a copy of currently selected text into a hidden buffer.

- cut — This deletes currently selected text from the screen and puts it in a hidden buffer.

- do it — This executes (runs) currently selected code.

- print it — This executes currently selected code like do it, but it also prints the result on the screen. print it is an appropriate command for the Smalltalk expression 3 + 4 whose result is, of course, 7. You see 7 displayed on the screen immediately after the 3 + 4 expression.

The Smalltalk-80 Language

Smalltalk is an object-oriented language. Things happen in a Smalltalk-80 system because objects are sent messages. Objects are system components which are described by their corresponding class. Each class contains a description of its object implementation. This description is used to create new instances of this class. Messages are requests for an object to carry out one of its operations. When you create an instance of a class you create an object that responds to a definite set of messages.

For example, suppose you need to simulate the behavior of traffic in a town. You might create classes of objects that represent traffic lights, cars, streets, and people. A traffic light would have three states through which it would cycle. Cars would stop and go along streets. People would cross streets and stop at traffic lights, etc. In the Smalltalk-80 system, you would design each object so that it responded to appropriate messages. For example, people objects and car objects might both respond to STOP and GO messages, but their response to STOP and GO messages would be different (we hope!). The Smalltalk-80 language is especially suited to problems solved by simulation techniques.

Objects and Classes

Since the Smalltalk-80 system and language is a completely object-oriented language and system, everything in the system can be thought of in terms of objects sending and receiving messages. Even the user interface — the windows, contents of the windows, the

commands in the pop-up menus, the mouse button actions, everything — is an object or a message being sent or received. (Because the Smalltalk-80 language gives you access to this, you can control almost every aspect of your personal Smalltalk-80 system.)

Objects are programming constructs that have some private memory and a set of operations that they can perform. So, you can think of objects as containing not only data structures but also as having access to subroutine-like algorithms, called methods. The methods are executed when one object receives a message that names a particular method, perhaps along with some arguments the receiving object needs to execute the invoked method.

You create an object by naming the class that describes it and by sending it an instance creation message (usually the new message). After this, you can invoke the methods that the object understands, that is, those which are in its repertoire of methods, by sending messages that denote its methods.

The Smalltalk-80 system consists of a large number (over 200) predefined classes of objects. These classes are arranged in a hierarchical order to facilitate the inheritance of the many methods that apply to them. For example, the classes Fraction and Float are subclasses of the class Number. Fraction and Float, since they are subclasses, inherit (and thus understand) all the methods that Number does. In addition, Fraction and Float understand methods that are unique to them that Number does not understand.

Methods and Messages

Methods are essentially implementations of algorithms. The execution of methods is how objects communicate with each other. This is how everything gets done in the system. You can think of methods as roughly equivalent to functions in the C language.

When you execute Smalltalk code, you send messages to objects by using the names of methods, called message selectors. If the object recognizes a specific message selector, it carries out the actions described by the method associated with that message selector.

Many message selectors must be accompanied by arguments. You can think of this as similar to the passing of parameters in subroutines. However, the arguments of message selectors are objects, and, because of the internal structure of objects, the analogy with parameter passing is incomplete. But this analogy does adequately express the basic idea.

Methods are shared among objects because of the hierarchical relationship of objects. Objects that are in subclasses understand all the methods known to objects in their superclasses. This is a powerful feature of a hierarchical object-oriented language like the Smalltalk-80 language.

The Syntax of the Smalltalk-80 Language

A good way to get the flavor of the Smalltalk-80 language is to take a look at some code from the system. (In Section 3, you are asked to find this code, run it, and make some changes to it. See Section 3 for a tutorial introduction to the System Browser, where you will find this code.)

An Example of Smalltalk Code

This example method (called example) for the class Pen, draws a square spiral figure on the screen in gray, moderately thick lines. (Refer to the tutorial to execute this code.)

Each line of this example is commented on below.

example

"Draws a spiral in gray with a pen that is 4 pixels wide."

```
| bic |
bic ← Pen new.
bic mask: Form gray.
bic defaultNib: 4.
bic combinationRule: Form under.
1 to: 50 do: [:i | bic go: i*4.  bic turn: 89]
```

"Pen example"

example

This is the name (or message selector) of the method.

"Draws a spiral in gray with a pen that is 4 pixels wide.""

This is a comment. You will find that there is a comment line included immediately following the message selector in many of the methods found in the Smalltalk-80 system. This initial comment usually gives a summary of what the method does. Comments are enclosed in double quotation marks and may be interspersed anywhere in the code.

| bic |

Two vertical bars (|) enclose the declaration of local variables. You can have more than one variable if necessary, each separated by a space from the next one. Here bic is the only local variable in the method. It is in good style to begin local identifiers with a lower case letter. Note that there is a blank line immediately preceding this line. The blank line separating the opening comment and the temporary variable declaration is also a matter of good Smalltalk style.

bic ← Pen new.

Pen is an object that behaves like a plotter pen. It understands direction, up, down, move, and so forth. Note that Pen begins with a capital letter. All class names in Smalltalk begin with a capital letter. new is the name of a message that is sent to Pen and it tells Pen to create an instance of itself. This newly created instance of Pen is then assigned — by the left arrow assignment operator — to the local variable bic. Note that the expression ends with a period. This separates well-formed expressions from one another.

(The left arrow (←) assignment operator is a non-ASCII character that gets mapped to the underscore character when a translation is done from Smalltalk code to pure ASCII characters. There is only one other special, non-ASCII symbol in the Smalltalk-80 system: the upward pointing return arrow, which does not appear in this example.)

bic mask: Form gray.

gray is a message that is sent to the object Form. Note that Form is capitalized. A Form object is a set of bits that represents a rectangular region for display on the Smalltalk screen. Since the Smalltalk-80 system supports a bitmap in which pixels are either black or white, shades of gray are simulated by patterns of black and white pixels. So, Form gray creates a gray (halftone) image. The mask: message specifies that a gray image be the mask for the Pen object bic. An instance of Pen understands the method denoted by mask: because mask: is already in its repertoire of methods.

bic defaultNib: 4.

Here 4 is an object that the message defaultNib: can take as an argument. bic, which is an instance of Pen, understands the message defaultNib: 4. This message tells bic to make its pen point four pixels in diameter.

bic combinationRule: Form under.

combinationRule: is a message that takes an integer as an argument. In the Smalltalk-80 system, integers are objects, so the expression following combinationRule: should evaluate to an integer. The expression Form under does indeed evaluate to an integer. In fact, it returns the integer 7. This is interpreted by bic as one of the sixteen combination rule modes that the drawing primitive BitBlt understands. BitBlt performs all drawing operations that are requested of Form objects. (Refer to the Goldberg and Robson book for more information about this.) The effect of this line of code is that the Pen object, bic, draws on the surface of the display and is not hidden or obscured by images that are there already, that is, it is ORed with the existing image.

1 to: 50 do: [:i | bic go: i*4. bic turn: 89]

This line of code should look reasonably suggestive to you if you know languages like C and Pascal. You should recognize it as an iteration loop. It says to evaluate the expression within the square brackets ([]) fifty times.

The part of the line enclosed in and including the square brackets is a block and it represents a deferred sequence of operations. The block has one argument specified by :i and separated from the expressions in the block by a vertical bar (|). There are two expression within this block: bic go: i*4 and bic turn: 89. bic go: i*4 means that the Pen object bic should draw in its current direction i*4 pixels. bic turn: 89 means to alter the current direction by 89 degrees.

"Pen example"

This is another comment because it is enclosed in double quotation marks, but look more closely at it. It actually forms a valid expression in Smalltalk code. The message selector example is the name of this particular method. Thus, the code within the quotes says to send the example message to the class Pen. Pen looks for a message selector named example, finds it, and executes the code you have just examined. (A commented Smalltalk expression at the end of example code is a common way to show how to execute the example code.)

This brief look at some Smalltalk code shows you how the primary programming constructs, objects and methods, have been implemented in a programming language. Pen is the name of a class that denotes a certain kind of object. An instance of Pen understands a definite set of messages and these messages denote methods (think of them as subroutines) that accomplish appropriate tasks for Pen objects. Note that all the individual terms denote either objects or messages. The Smalltalk-80 system is completely objected-oriented. Sometimes this seems odd when you are asked to think of

3 + 4 as: send the object 3 the message + 4. On the other hand, bic defaultNib: 4, meaning send the Pen object, bic, the message to make its point four pixels wide, seems quite understandable and natural on first meeting. Just remember that everything you deal with in the Smalltalk-80 system is an object.

SECTION 3

THE 4404 SMALLTALK-80 SYSTEM: A TUTORIAL

This section is intended for the first-time user of the Smalltalk-80 system on the 4404. Its purpose is to get you into the Smalltalk-80 system, show you some commonly-used features, and exit you back to the 4404 operating system. The information in this section is presented as a tutorial, which means that you should turn on your 4404 and actually perform the steps on your own 4404 as you go through the tutorial. In the Smalltalk-80 system, there is no substitute for actual practice, especially concerning the behavior of the mouse, windows, and menus.

The Goldberg book (referred to in Section 1) is a very thorough introduction to the Smalltalk-80 system, and you will find that the 4404 version of the Smalltalk-80 system is very nearly the same as that in the Goldberg book. You can use the tutorial in this section to get a first acquaintance feel for the Smalltalk-80 system, especially as it relates to the 4404. Then, after you go through the tutorial, you can turn to the Goldberg book to continue your introduction to the Smalltalk-80 system.

How to Get the Most Out of This Tutorial

This tutorial is intended for programmers who are new to the Smalltalk-80 system. If you are already acquainted with the Smalltalk-80 system on another machine, you should turn to Section 4. It gets you quickly into the Smalltalk-80 system.

The Smalltalk-80 system depends on the 4404 electronics and the 4404 operating system. Make sure that you are sufficiently familiar with these components before you go on to work with the Smalltalk-80 system. You can become familiar with these components by reading through the 4404 AIS User's Manual. Probably half an hour or so with the User's manual is sufficient.

Tutorial on the 4044 Smalltalk-80 System

This tutorial assumes that you are performing the steps on your 4404 in the order that they are given. As you go through the tutorial and feel more confident about how the mouse, windows, and menus work, feel free to try out operations similar to the ones described. However, in the beginning, it is easy to miss some subtlety of mouse/window/menu interaction that could end up getting you "lost" in the Smalltalk-80 system. So, in the beginning, an attitude of cautious exploration is urged.

You should allow around two hours to go through the tutorial.

Getting Into the Smalltalk-80 System

1. Make sure all cables and power cords are connected and plugged in properly. (Refer to the 4404 AIS User's Manual if you are unsure of this.)

2. Turn on the Mass Storage Unit (MSU). (In the standard 4404, this contains the hard disk drive and one flexible disk drive. The on/off button is in the lower center front of the MSU.)

3. Turn on the CPU/Display Unit. (The square, on/off button for the CPU/Display Unit is in the lower righthand corner.)

4. You should hear hard disk activity and see the activity light flicker as the 4404 operating system is loaded into main memory. (The activity light is in the upper righthand corner of the MSU.) After a few seconds, you should see this operating system prompt†:

   ++

5. At the prompt ++, type:

   smalltalk

   followed by a carriage return.

6. You should hear more hard disk activity as the Smalltalk-80 system files are loaded into main memory. These files are very large so be prepared to wait approximately forty seconds as they load. You will see the Smalltalk interpreter sign-on message appear fairly soon, after which the interpreter loads in the Smalltalk-80 image file.

   While you are waiting for the Smalltalk-80 system to load, make sure that the Mouse Pad is secured on a flat, level surface to the right or left of the Display/CPU unit. Set the mouse on the pad with the cord leading away from you. This orients the three mouse buttons away from you. This is the correct orientation for the mouse.

Learning Mouse Mechanics

7. You should now see the initial Smalltalk display on the screen. See Figure 1. You will note that there are a number of boxes with text and Smalltalk code in them. The boxes and parts of boxes are window borders and panes within the window borders.

8. Grasp the mouse and, while keeping it flat against the mouse pad, move it back and forth and up and down around the pad. Observe that the black arrow cursor moves in concert with your motions on the pad. The tip of the arrow is the pointing part of the arrow cursor.

9. Note that, as you moved the mouse over one of the windows, one of the small title boxes at the upper left corner of one of the three visible windows became highlighted, that is, turned to white letters on a black background. You have just

---

† If a password has been installed on your system, you must "login" to see this prompt. See the 4404 AIS User's Manual for more information.
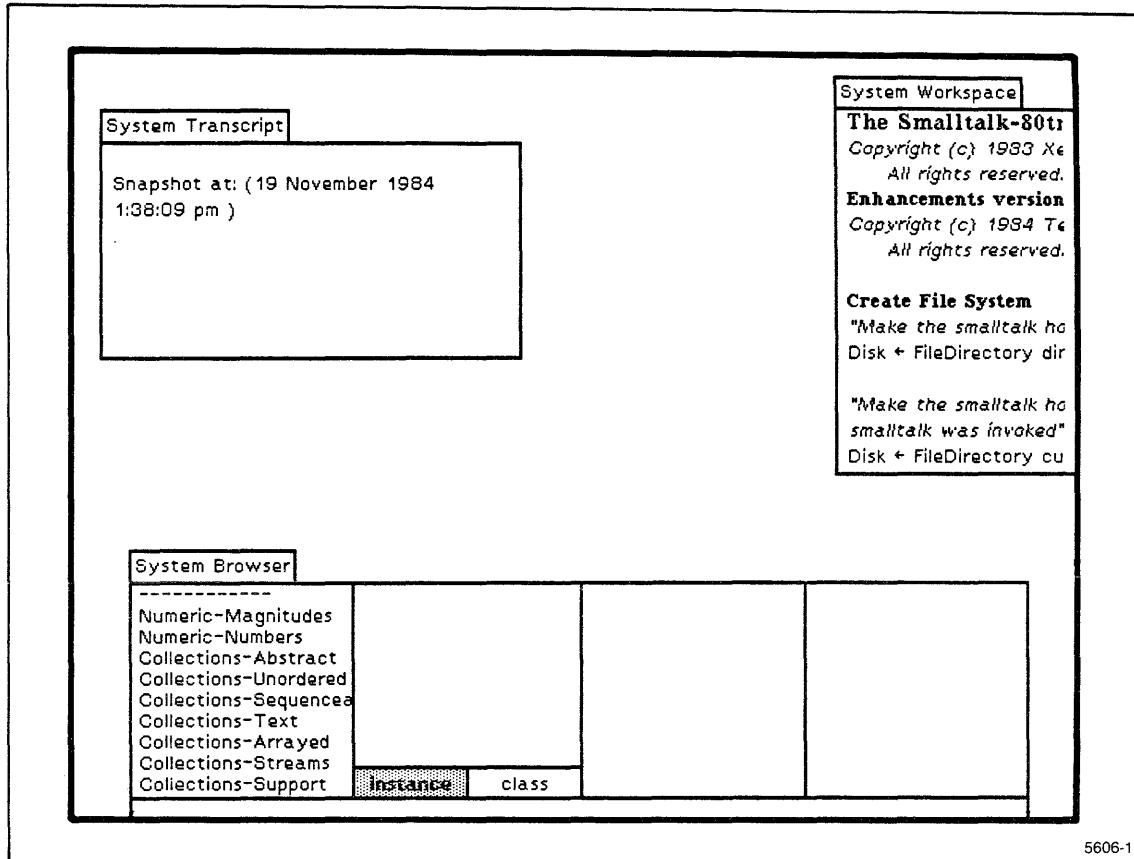
Figure 1. Initial Screen of Standard Image.

selected that window to be the active window. (There is always only one active window.)

10. Now move the mouse so that the arrow cursor moves up against the right side of the display. Watch the whole display move as you continue to move the mouse in the same direction. While doing this, you may reach the end of the mouse pad. Just pick up the mouse and move it back to the other end of the pad and continue the movement in the same direction.

11. Now move the mouse so that the arrow cursor moves down against the bottom of the display. Watch the whole display move as you continue to move the mouse. Again pick up the mouse and replace it on the pad as you need to, to continue the movement.

The joydisk in the upper lefthand corner of the keyboard is another way to move the whole display at once. What you are observing is a 640 by 480 sized pixel window into the Smalltalk-80 system's 1024 by 1024 bitmap display. This "hardware" panning is completely independent of the Smalltalk-80 system and may be done at any time.

What the Mouse Buttons Do

12. The three buttons on the mouse give you access to many of the Smalltalk-80 system's functions. The left button is used mainly to select something: some text, some code, a window, etc. The middle button is used mainly to pop up a menu dealing with what you can do to the contents of windows. And the right button is used mainly to pop up a menu dealing with what you can do to a window itself.

    A good way to manipulate the mouse is to grasp it between your thumb and middle finger and, then, to press the buttons with your index finger. After you work with the mouse for a while, you may find that another way works better for you. The point of the mouse is to facilitate quick interaction with the Smalltalk-80 system; use the mouse the way it works best for you.

    Each button has two states: pressed and released. When you work with the mouse, be ready to press and hold a button down. You release the button after you have decided (with the middle and right buttons) what command you want to do. The general rule is: press, hold, decide, release! (Later in the tutorial you will encounter the phrase to "click" a mouse button. This means to press and immediately release a button. Clicking is used less often than pressing, holding, deciding, and releasing in the Smalltalk-80 system.)

How to Select Smalltalk Objects

13. Move the arrow cursor to the inside of the *System Transcript* window. (Remember that the tip of the arrow is the active selection point of the arrow cursor.) Press and immediately release the left mouse button. Note that the window's title box turns to black showing you that the window is selected and active. Left button activity anywhere in a window, including its title box, activates that window.

14. Move the arrow cursor to the System Transcript window and activate it (press and release the left mouse button). Note that, in addition to some English text, there is a small dark caret cursor. This is the point at which typed characters appear in the text. Type a few characters. Take up the mouse again and move the arrow cursor between two characters along a line of text. Press and release the left mouse button. Note that the caret cursor has moved to that point. Type a few more characters. If you like, move the caret to the end of the last line of text, type a few carriage returns and type a sentence or two and watch the text automatically wrap around.

15. Press and hold down the left mouse button while you move the arrow cursor around the System Transcript window. Note that parts of text are highlighted and "unhighlighted" as you move. Release the button, move the arrow cursor to a new initial location, press and hold down the left mouse button again, and move the arrow cursor around. Do this a few times till you get a feeling for how highlighting works. Note that the arrow cursor remembers an initial position. Now go to the line just above or below all of the text (or to the very beginning of all text in the window, if there is no blank line at the beginning). Press and release twice (or three times). Note that the entire text is now highlighted. The text or code that is highlighted is selected and ready to be manipulated, usually by applying a command to it.

16. Deselect ("unhighlight") all of the text in the System Transcript window. Pick out a word in the text, move the arrow cursor point just before the word, press and hold the left button, move the arrow cursor to the right along the line, and, at the end of the word, release the button. You should see your selected word highlighted. Now type another word. You should see your word replacing the previously highlighted word. This is one way to replace text or code in a window. In fact, typing always inserts new text and replaces the current text selection. This is even true of the caret text position cursor if you think of it as a zero length selection. There is nothing to replace, in this case, but it does insert whatever you type.

How to Scroll Text in a Window

17. Now move to the System Workspace window and press and release the left mouse button. See Figure 2. This selects the System Workspace window to be the active window. Move the arrow cursor into the small box attached to the side of the main window box. This is the scroll bar region. In windows that have a lot of text, you use this feature to move the text up or down in the window. There are three basic ways to use the scroll bar: you can move text up one or more lines at a time, move it down one or more lines at a time, or move yourself quickly to some part of the whole text. The gray scroll bar's length (compared to the whole scroll box) shows you approximately what proportion of the entire text is currently in the window. The gray scroll bar also shows you (by where it is in the scroll box) whether you are at the top, middle, or bottom of the entire text. For example, if you see the gray scroll bar relatively small at the top of the scroll box, then you know that you are looking at the very beginning of a lot of text.
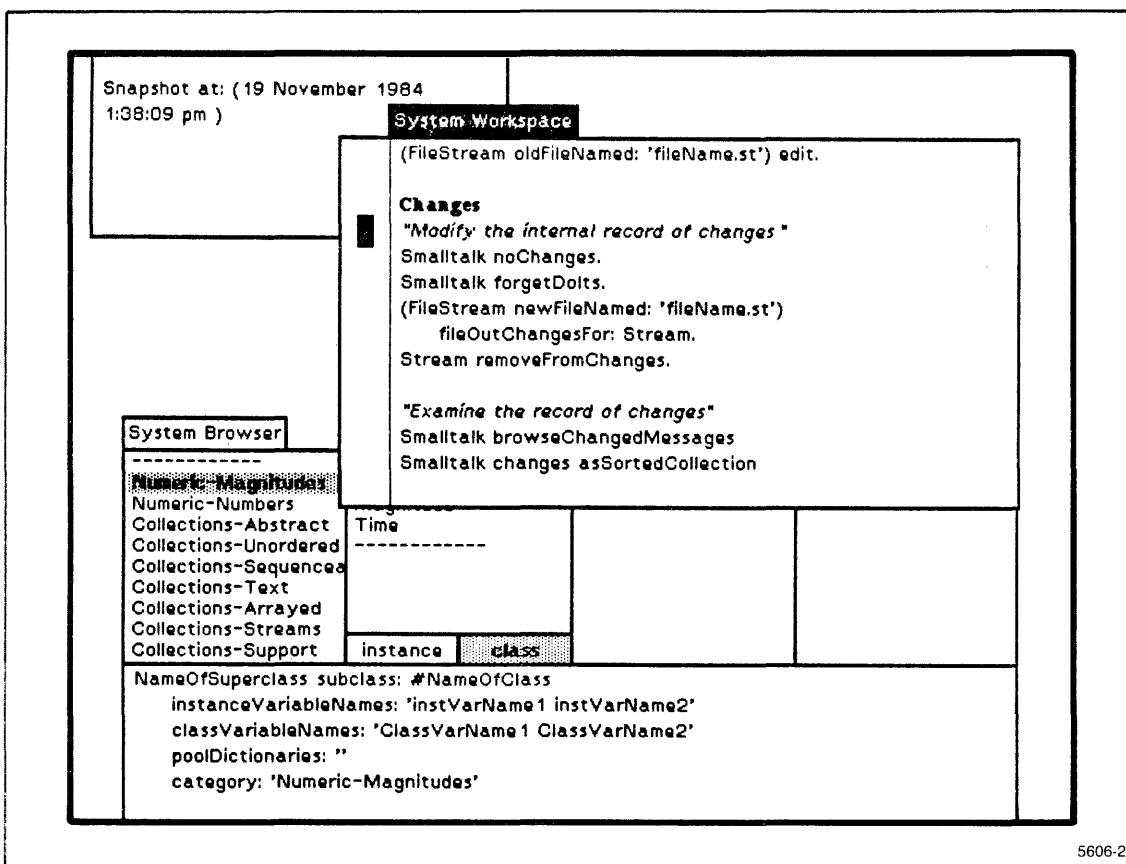
Note that as you move the arrow cursor horizontally across the scroll box three new cursors appear: a downward-pointing half-arrow, a horizontally-pointing arrow, and an upward-pointing half-arrow. Find the downward-pointing half-arrow and press and release the left mouse button. Observe that the text moves up one or more lines for each press and release, depending on whether you are at the top, middle, or bottom of the scroll box. The farther down you go in the scroll box, the greater the number of lines you scroll at a time. The downward-pointing half-arrow works analogously. Now find the horizontally-pointing arrow, press and hold the left mouse button, and move the mouse up or down. Observe that the gray scroll bar follows your motion. Release the left mouse button with the scroll bar positioned at the bottom of its box. You are now looking at the end of the text in the window. Now move the scroll bar to the top of the box. You are now, of course, looking at the beginning of the text in the window. Play with the scroll bar cursors until you feel comfortable with their operation.

How to Find Out About Classes and Messages

18. At this point, you have learned the basic operation of the mouse and left mouse button. You can now explore the middle and right mouse button operation and learn something about the System Browser at the same time.

Move the arrow cursor into the System Browser window and click (press and immediately release) the left mouse button. You have, of course, just activated the System Browser. See Figure 3. Note that it has five major panes: four small ones at the top and one large one below these four. As you can see, the leftmost small pane has some class category names in it. Move the arrow cursor into this small pane and select by clicking on the line with the left mouse button the class category

```
Snapshot at: (19 November 1984
1:38:09 pm )
                          System Workspace

                          (FileStream oldFileNamed: 'fileName.st') edit.

                          Changes
                          "Modify the internal record of changes "
                          Smalltalk noChanges.
                          Smalltalk forgetDoIts.
                          (FileStream newFileNamed: 'fileName.st')
                              fileOutChangesFor: Stream.
                          Stream removeFromChanges.

                          "Examine the record of changes"
    System Browser        Smalltalk browseChangedMessages
    -----------           Smalltalk changes asSortedCollection
    Numeric-Magnitudes
    Numeric-Numbers
    Collections-Abstract  Time
    Collections-Unordered -----------
    Collections-Sequenced
    Collections-Text
    Collections-Arrayed
    Collections-Streams
    Collections-Support   instance   class

    NameOfSuperclass subclass: #NameOfClass
        instanceVariableNames: 'instVarName1 instVarName2'
        classVariableNames: 'ClassVarName1 ClassVarName2'
        poolDictionaries: ''
        category: 'Numeric-Magnitudes'

                                                          5606-2
```

Figure 2. System Workspace Window.

Numeric-Numbers. You should see *Numeric-Numbers* highlight, if you have successfully selected it. Note that the next small pane over now has a list of Smalltalk class names in it. (In contrast to the class categories, the class names are actual Smalltalk-80 language expressions.) And note also that the bottom pane has what is called Smalltalk template code. (This is code that you edit and then execute.)

19. Move the arrow cursor to the class names pane. (This one has the two boxes at the bottom with *instance* and *class*; make sure *instance* is highlighted by clicking with the left button on *instance*.) Select the class name Integer. Now the next pane to the right fills with a list of message categories and the bottom pane template code changes again. Select **factorization and divisibility** in the message category pane. Again the next pane over, the message selector pane, fills with a list of expressions. These are actual Smalltalk code message selectors. You will find these throughout Smalltalk code as you examine the code in the system. Finally, go to the rightmost pane and select factorial (asterisk). This symbol is the message selector for factorial.

20. You now see in the bottom pane the actual Smalltalk code that is executed if you compile and run a Smalltalk expression such as **4 factorial**. After you have examined the code to your satisfaction, try going back to the message selector pane and

Figure 3. System Browser Window.

clicking on gcd: or lcm:. At this point, if you like, you can go back to the message category pane, click with the left button on a different message category, and then, click on any new message selector you see in the message selector pane. You use this feature of the System Browser constantly as you learn about the code in the system.

Running Some Code Already in the System

21. If you have read through Section 2, you will remember that a method whose message selector is example was examined. Now use the System Browser to find and run that piece of code. Go to the class category pane (the leftmost one) and click with the left button on the category *Graphics-Primitives*. (You may have to use the scroll bar to find the category in the list.) Next click on the class Pen. Be sure to also click on the *class* box in the class names pane. Then click on *examples* in the message categories pane and example in the message selectors pane. In the bottom pane, you should now be looking at the example method for the class Pen. See Figure 3.

22. To run this example of Smalltalk code, look to the end of the code and find the text Pen example. Highlight just the Smalltalk expression Pen example, not "Pen example", that is, do not include the double quotation marks since this would make
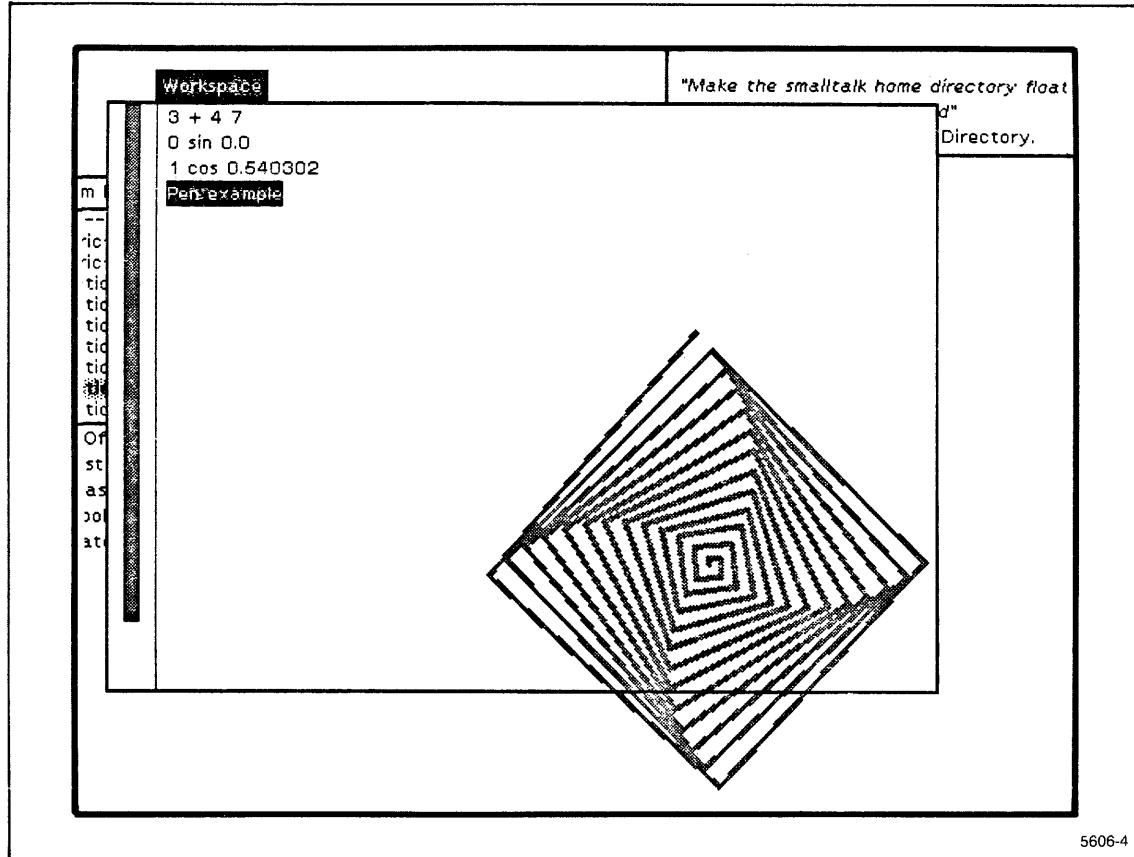
Figure 4. Workspace Window.

the expression into a comment. There are two ways to do this: (1) put the arrow cursor point between the initial double quotation mark and the P, press and hold the left mouse button, run the cursor to the right till you reach the last e, and release the button; (2) put the arrow cursor between the double quotation mark and the P and click the left mouse button twice without moving the mouse. (For the second method to work, you must place the arrow cursor point immediately after the double quotation mark if there are any spaces between the double quotation mark and the first character on the line.)

23. With just Pen example highlighted, press and hold the middle mouse button. Observe that a pop-up menu appears overlaying the window information. Continue to hold down the middle mouse button while you move the arrow cursor up and down the menu. You see a number of commands that you can invoke. Here again, you invoke whatever command is highlighted when you release the middle mouse button. To run the Pen example method, choose the command *do it*. You should see a square spiral shape appear on the display. Also, note that, as the method is executed, the arrow cursor changes to an arrow cursor with a star. This appears when the Smalltalk-80 system is executing code. Choose *do it* when you want to execute Smalltalk code.

(A good technique to use if you have a menu showing and have decided to choose none of the commands is to simply slide the cursor to the side, out of the menu and release the button.)

Clearing the Display

24. There are many times when the display gets cluttered with things that you are working with like the square spiral. At any time, you can restore the display with a middle mouse button pop-up menu. To see this menu, you must be outside all windows, that is, the cursor must be positioned on the gray background. Move the cursor to the gray background and press and hold the middle mouse button. Observe that there are a number of menu choices; for now, go to the very top of the menu, highlight *restore display*, and then release. You will see the entire display being rewritten. Note that the square spiral disappeared; it is indeed gone since it was not saved.

Altering and Running Some Smalltalk Code

25. By examining the Smalltalk Pen example method code, you will probably be able to pick out certain constants that affect the shape and tone of the figure. Use the left mouse button to highlight the 4 in the expression bic defaultNib: 4.. Now type 6. You should see the expression change to bic defaultNib: 6. Once you are sure you have changed just the 4 to a 6 and nothing else in the code, choose the middle mouse button and press and hold it. Find the command *accept* in the pop-up menu. (It is below *do it*.) Select *accept* by highlighting it and releasing the middle mouse button. You should observe that the Smalltalk busy signal — the arrow cursor accompanied by the star — appears for a few seconds and then returns to the "ready-for-input", plain arrow cursor. You have just recompiled the example method, and, of course, this had to be done to enter the altered code into the Smalltalk-80 system. You are now ready to *do it* and observe the result of your change. Do not forget to select Pen example first, so that *do it* has something to operate on. Do this now: select Pen example and then *do it*. You should see now that the lines of the figure are thicker than before.

    If you like, you can change the gray lines to black. Select gray in the expression bic mask: Form gray. Change gray to black. Select *accept* from the middle mouse button menu to recompile the method. Next, select Pen example and then *do it*. You should observe that the figure changed from gray to black lines.

    If you feel really adventurous, you might try changing some of the other numerical constants that you see in the method code. If the code does not *accept* properly, an error message is inserted into the method and highlighted or you will see a window pop up that attempts to inform you about what the problem is. If you do not understand either these (a likelihood at this point), choose the middle button menu selection *abort* (or the right button menu selection *close*), and try again to change the code.

Take a Short Break

26. At this point, you might like to take a short break and go through the rest of the tutorial later. If it is at all possible, leave the system just as it is now. (Note that the display blanks out after about 10 minutes of no activity at the keyboard or mouse. If this happens, just move the mouse or press the shift key. The display

reappears just as you left it.) If you cannot do this, go to the last step in this tutorial (step 34) and *quit* the system, then, when you come back, go to the beginning of the tutorial and repeat steps 1 through 6, then skip back to here.

Opening a Workspace Window

27. If you have not done so recently, clean up the screen by placing the arrow cursor on the gray background and selecting *restore display* with the middle mouse button. Now go back to the middle mouse button pop-up menu and select *workspace* to open a workspace window. A small square should appear along with an angle bracket at the upper left corner. You can drag this square anywhere over the display you like, even over already existing windows. When you have the square where you want it, depress and hold the left mouse button. You will see that the angle bracket has shifted to the lower right corner. Now stretch the square out by moving diagonally away from the upper left corner. When the window fills about a third or so of the display, release the left mouse button. You will note that the depression of the left mouse button anchors the upper left corner and the release fixes the size of the window. See Figure 4.

Evaluating Smalltalk Expressions in a Workspace

28. Workspace windows are where you usually develop Smalltalk code. Place the arrow cursor within the workspace window and click the left mouse button. Set the arrow cursor to one side within the window. Type these Smalltalk expressions in the workspace:

> 3 + 4
> 0 sin
> 1 cos
> Pen example

First, highlight the expression 3 + 4. Now choose the middle mouse button menu command *print it*. Note that 7 appears just to the right of the expression. Any time you want to find out what a Smalltalk expression evaluates to, you can choose *print it*. Now select first 0 sin and *print it* and then 1 cos and *print it*. The last expression, Pen example, is by now quite familiar to you. Select it, but this time select the *do it* command from the middle mouse button menu. Observe that the Smalltalk-80 system understands the expression and executes it just as it did in the System Browser.

Manipulating Code or Text in a Workspace

29. To facilitate common text manipulation tasks, the Smalltalk-80 system has implemented the commands *undo, copy, cut,* and *paste* in the middle mouse button pop-up menu. Highlight the expression Pen example in the workspace. Select the middle mouse button command *cut*. It does what you would expect; it removes the text that you highlighted. It also does what you might expect: it puts the text in a temporary buffer in memory. Now choose the middle mouse button command *undo*. You should see the expression reappear.

Try this. Highlight Pen example as before. But now select *copy* from the middle mouse button menu. You have just copied the text Pen example into the buffer in memory. Now move the arrow cursor into the gray background and press and hold

the middle mouse button. Choose the *workspace* command and open a workspace window. With the arrow cursor in the workspace window, choose *paste* from the middle mouse button menu. You should see the text Pen example appear in the window. You may, of course, copy and paste virtually any size block of text you like. *copy* and *paste* allow you to very easily shift blocks of text between windows. (If you would like to get rid of the workspace window after you are done with it, choose the right mouse button command *close*.)

Communicating with the 4404 File System

30. After you have worked with the Smalltalk-80 system for a short while you will undoubtedly want to communicate with the 4404 operating system to get a directory of files, look at the contents of files, output text or code in a window to a file, and get the contents of a file for use in the Smalltalk-80 system. To do these things, you need to open a File List window.

    In opening a File List window, the action that occurs is analogous to opening a workspace window. Make this File List window about half the size of the visible screen. (You can put the window right over the System Browser and System Workspace windows if you like.) Do this now. Move the arrow cursor to the gray background and choose the command *file list* from the middle mouse button menu. Now move the window where you want it, press and hold the left mouse button to anchor the upper left corner of the window, move the lower right corner to where you want it, and then release the button. See Figure 5.

Listing Files

31. To get a list of the /smalltalk directory's files, go to the top pane of the window (just under the title box) and type */smalltalk/*. The asterisk (*), the wildcard character, stands for all possible character combinations. Now, with the cursor still inside the top pane, press the middle mouse button and choose *accept*. You should see the middle pane of the window fill with the names of files in the /smalltalk directory. Use the scroll bar to find the file *standardImage*. Select this file with the left mouse button. In the bottom pane, you should see a message about the size of the file and the time it was created. It should look something like this:

    758256 bytes
    23 September 1984
    12:12:07 pm

    Select some other files and look at the information in the bottom pane. Some of the files are directories. Choose the file name *system*; this should be a directory. Choose the middle mouse button menu command *list contents*. Observe that the bottom pane now contains a list of the files in that directory. If you would like to look at more than the names of the files, go to the middle mouse button menu and choose the command *spawn*. You will note that a second File List window opens. When you are through looking at these files, choose the right mouse button menu command *close* and go back to the original File List window.
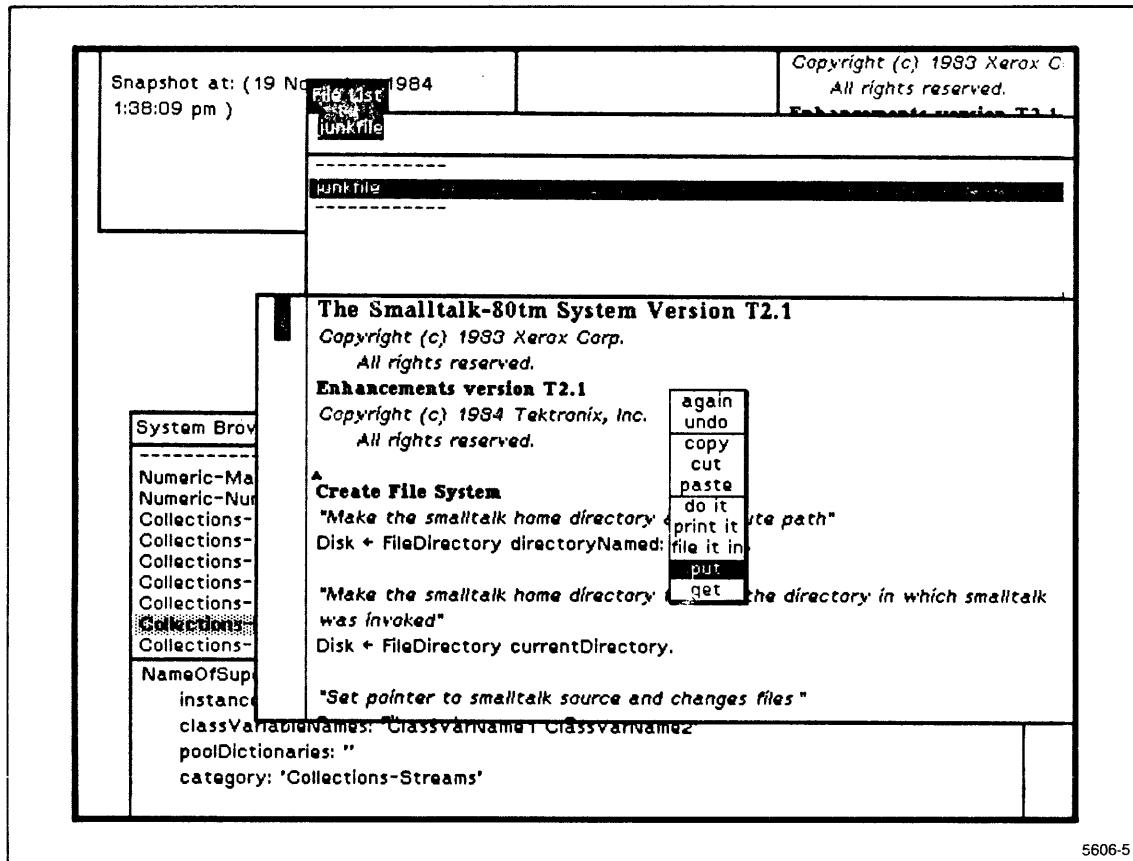
Figure 5. File List Window.

Writing Files Out to the 4404 Operating System

32. Suppose now that you would like to create a new file and put some text from the Smalltalk-80 system into the file. Here is how you do that:

   A. First, open a File List window. In the top pane, type in the name of the file you want to put the text into. Here type in *junkfile*. (Be sure to *cut* the asterisk or any other characters that might be in the top pane, so that you have just the file name *junkfile* there.) Select the middle mouse button menu command *accept*. You should see *junkfile* appear in the middle pane. Now select *junkfile* in the middle pane with the left mouse button. You should see the message *-new file or directory-* in the bottom pane. If you do not, try another file name until you can be sure you have a truly new file name.

   B. Select the *-new file or directory-* message and *cut* it from the window. Now go to the System Workspace window and select all of the text in that window by going to the very beginning of the text and clicking until you see all of the window text highlighted. Do a *copy* command from the middle mouse button menu.

C. Go back to the bottom pane of the File List window and do a *paste* from the middle mouse button menu. The entire text from the System Workspace window should now be available to you in the File List window. Use the scroll bar and verify that this did indeed occur.

D. While you are still in the bottom pane of the File List window, press the middle mouse button and look to the bottom of the pop-up menu. Choose the command *put*. You should hear some disk activity as the Smalltalk-80 system outputs the contents of the File List window into the file called *junkfile*.

Now, to verify that the file did get written, go to the middle pane and deselect *junkfile* by clicking on it. Now select *junkfile* again and the bottom pane of the File List window should contain information about the size of the file. To inspect the contents of the file, go to the middle pane and use the middle mouse button to do a *get contents*. Now you should see the contents of junkfile reappear in the bottom pane of the File List window.

Manipulating Windows Themselves

33. The right mouse button menu normally has commands that apply to windows themselves in contrast to the contents of windows. Choose a window on the display and activate it by clicking with the left mouse button. Now, press and hold the right mouse button. You will see this menu:

title
under
move
frame
collapse
repaint
close

- *title* allows you to change the title of a window.

- *under* allows you to bring a window into view if it is under another window. (Windows overlapping each other are arranged in an internal stack. *under* pops the window at the arrow cursor point on the bottom of the stack to the top.)

- *move* allows you to change the position of a window on the display.

- *frame* allows you to change the size of a window.

- *collapse* allows you to delete from the screen all of a window except its title block.

- *repaint* allows you to redisplay the contents of the current window.

- *close* allows you to remove a window entirely from the display (and discard all of the contents of the window too).

Take the time now to work with these commands until you feel reasonably comfortable with their operation.

Saving Your Work and Quiting

34. After you are through with this tutorial session, you will probably want to quit without saving your work. To do this, use the middle mouse button menu command *quit*. If you are ready to quit, do this now. You will see a second window pop up. As you can see, you have three choices. For now, choose *Quit, without saving*.

    However, in later sessions, you will undoubtedly want to save your work. Then you should choose *Save, then quit*. You will see a prompter window pop up with a default file name. You can choose the default name by *accept*ing it or entering a carriage return. Or you can enter your own file name. If you like, save the image file as */public/myImage* or enter your own image file name. At this time, the system makes what is called a "snapshot" of your current image. When you wish to take up exactly where you left off with your work, just invoke your own snapshot image file, *myImage* by typing *smalltalk myImage* at the operating system prompt.

You have now finished this tutorial. Congratulations! You should now be able to reinforce what you have learned here and extend your knowledge of the Smalltalk-80 system by going directly to the Addison-Wesley Smalltalk books. (See Section 1 for more information about these books.) Please note that the last two sections of this manual are meant mainly for the person who has already absorbed what is in the Addison-Wesley books.

SECTION 4

PROGRAMMING IN THE SMALLTALK-80 SYSTEM

This section is intended for the experienced Smalltalk programmer. That is, it is assumed that you have actually done some Smalltalk programming on another machine. If you have not, you should probably read through Sections 2 and 3 first, the Addison-Wesley books next, and then come back to this section.

This section presents three topics. First, it tells you how to get into the Smalltalk-80 system quickly. Second, it presents a number of programmer hints, helps, tips, etc., to make your life as a Smalltalk programmer easier. And, third it explores a topic that is not extensively treated in the Addison-Wesley "Goldberg" or "Goldberg and Robson" books. This is Model-View-Controller. Some understanding of this topic is necessary to write Smalltalk applications involving the window/user interface.

Getting into the Smalltalk-80 System on the 4404

The Smalltalk-80 system depends on the 4404 machine itself and the 4404 operating system. Make sure that you are sufficiently familiar with these components before you go on to work with the Smalltalk-80 system. You can become familiar with these components by reading through the 4404 AIS User's Manual. Probably half an hour or so with this manual is sufficient.

The standard Smalltalk-80 system consists of four files as your 4404 is configured at the factory. They are:

- *smalltalk*   This is the Smalltalk interpreter for the 4404. This file is found in the directory */bin*.

- *standardChanges*   This is the file that receives any changes you make to the standard system, and it is found in the directory */smalltalk/system*. This file tends to grow each time you use the standardImage.

- *standardImage*   This is the default Smalltalk-80 virtual image file, and it is found in the directory */smalltalk*.

- *standardSources*   This file contains the source code for each of the compiled methods in the standardImage file. This file is found in the */smalltalk/system* directory.

The Smalltalk interpreter is invoked by typing the *smalltalk* command. This command has a single optional parameter that is the name of a Smalltalk-80 virtual image file. If the parameter is omitted, the interpreter loads the standard virtual image named */smalltalk/standardImage*.

After you have created your own virtual image file by using the *save* command, you can load it instead of the standardImage. If you are not in the same directory as your image, you need to type the complete path name of your image as the argument to the *smalltalk* command.

Invoking the Smalltalk-80 system for the First Time

1. Make sure all cables and power cords are connected and plugged in properly. (Refer to the 4404 AIS User's Manual if you are unsure of this.)

2. Turn on the Mass Storage Unit (MSU). (In the standard 4404, this contains the hard disk drive and one flexible disk drive. The on/off button is in the lower center front of the MSU.)

3. Turn on the CPU/Display Unit. (The square, on/off button for the CPU/Display Unit is in the lower righthand corner.)

4. You should hear hard disk activity and see the activity light flicker as the 4404 operating system is loaded into main memory. (The activity light is in the upper righthand corner of the MSU.) After a few seconds, you should see this operating system prompt:

    ++

5. At the prompt ++, type:

    smalltalk

    followed by a carriage return.

6. You should hear more hard disk activity as the Smalltalk interpreter and the standard 4404 Smalltalk-80 virtual image are loaded into main memory. The virtual image is a large file so be prepared to wait approximately forty seconds as it loads.

7. You should now see the 4404 screen displaying a familiar sight: the Smalltalk System Browser window, the System Transcript window, etc., of the Smalltalk-80 system. Note that, on the 4404, you see a portion of the entire Smalltalk-80 system screen. You can look at other parts of the screen two ways. If you move the mouse cursor to the edge of the screen, the screen does a hardware scroll and brings into view previously hidden parts of the entire screen. Alternatively, you can scroll the screen with the joydisk (the octagonal-shaped disk in the upper lefthand corner of the keyboard). Note that, with the joydisk, you can pan the cursor off the visible display. Press the Cursor Center Key, F12, to bring the cursor to the center of the visible display if you need to.

If the sequence of events described in this procedure does not occur as you expect, you should refer to the 4404 AIS User's Manual.

Installing Your Own Image

After you have had enough experience with the editor and menus to feel comfortable editing and executing Smalltalk code, you may wish to install your own personal image. To do this, follow these steps:

1. **Establish your Disk directory.**
   Disk is a global variable representing an instance of a FileDirectory. Filenames that do not begin with a '/' are assumed to be relative to this directory. For instance, doing a *file out* in a Browser creates a file in the Disk directory. You should have write access to this directory.

   Use the first template in the System Workspace under the *Create File System* heading to set up your Disk directory. Edit the template, select the entire line, and then execute it with the middle button menu item *do it*. For example, your Disk might be:

   Disk ← FileDirectory directoryNamed: '/public'.

   The directory should previously exist.

2. **Install your changes file.**
   Find the line in the System Workspace (several lines down from the expression you executed to initialize your Disk directory) that begins with:

   SourceFiles at: 2 put:

   Edit the next line so that it looks something like this:

   (FileStream oldFileNamed: 'changes').

   Each user should have a separate changes file. (You may use a different name for your changes file). The first time you execute these two lines with a *do it* command, a notifier should tell you that the file does not exist. *Proceed* so that the file is created. If you do not get a notifier, someone else has already created a file with this name. You should choose a new name for your changes file and re-execute these two lines. Every time you execute a *do it* or *accept* a new method, that action is recorded in your changes file.

3. **Save your templates.**
   Use the middle mouse button *accept* command inside the System Workspace. This saves the contents so that all new System Workspaces opened from the System Menu reflect an up-to-date version of the contents of the System Workspace.

4. **Make a snapshot.**
   Use the System Menu to make a snapshot. You then are prompted for the name of a file that receives your personal copy of the Smalltalk-80 virtual image. If you type a path name that begins with a slash, the image is saved in a file with this complete path name. If the name is not a complete path name, it is used as a path name relative to Disk. For example, if Disk is a FileDirectory on */public/sam* and the image name *myImage* is given, the image is written to the file */public/sam/myImage*.

   Invoke the Smalltalk interpreter with the image name as a parameter to load the image saved in that file.

Hints, Helps, and Tips for Smalltalk Programmers

The topics under this heading have been suggested by knowledgeable Smalltalk programmers. Generally speaking, these hints, helps, and tips are intended to make commonly performed tasks easier to do.

Managing Your Image and Changes Files

Since the Smalltalk-80 system allows you complete freedom to personalize and change the Smalltalk programming environment to suit your needs, there exists the possibility that you may make modifications that render your system inoperable. If you are going to be programming at this level, the following suggestions may help.

A good practice to follow is to alternate between two image files as you make changes to your system, that is, snapshot alternately between the two files. This way you are always able to go back to a previous image.

Another recommended strategy is to record your modifications in an external file as Smalltalk source code. This allows you to recreate all modifications to your image by filing in the source code. To do this, locate the template (FileStream newFileNamed: 'filename.st') fileOutChanges in the System Workspace. Modify the string representing the file name, then select and execute this expression. Executing this template records modifications that are contained in the ChangeSet for the current project. A ChangeSet is an internal list of modifications, one per project. For instance, you might create two projects for two distinct applications in your Smalltalk-80 image. Since each project represents another screenful of information as well as another ChangeSet, this is a convenient way to organize your work. As long as changes relevant to each application are made in their respective projects, the template mentioned above records modifications for that project. Chapter 4 of the Goldberg book contains more information on projects.

Note that it is possible to modify the current ChangeSet. See Chapter 23 of the Goldberg book for more information on the ChangeSet, how it can be modified, and how it can be used to recover from crashes.

Modifying your Changes File

As you can see from the discussion above, ChangeSets are a means of keeping a list of modifications that are recorded in your changes file. The facility to condense a changes file is provided because much of the time modifications to a method go through several iterations before a final version is decided upon. Each time the method is recompiled, it is appended to the changes file, even though only one version is referenced. In the System Workspace under the heading *Changes*, execution of the expression Smalltalk condenseChanges renames the original changes file to a backup name and creates a new changes file with duplicates and doIts removed. Be sure to save your image after condensing changes, so that the references to the new changes file are saved.

The section entitled "Creating a New System Image" beginning on page 478 of the Goldberg book discusses how and why to clone an image. Since the format of Xerox and Tektronix images is different, you must evaluate the expression TekSystemTracer writeClone to create a cloned image that can be brought up on the 4404. Circular garbage is reclaimed as a part of the normal garbage collection process, so there is no need to clone an image to release these objects in a 4404 image. You may still want to make a cloned image to release old Symbol table entries and, since a clone image is

completely re-organized, to possibly obtain better locality of reference, which means that the image might page less.

If You Cannot Bring Up Your Image...

After you have created your own image and saved some of your work in your own image file, you may have created an image that does not load properly. If this happens, here are a few suggestions to help you track down the problem.

1. Use the 4404 operating system command *dir +l* to check the size of your image file. It should be about the same size as the standard image file. The standard image file, *standardImage*, is approximately 1500 blocks in size.

2. If your image file seems to be about the right size, check to make sure that you have enough free memory on the disk for the operating system to function. Remember that the operating system uses part of the winchester disk space as main memory because it uses a virtual memory paging scheme to extend main memory. Use the operating system command *free /dev/disk* to find out how much memory is left on the disk. To be safe, you should have at least 1 megabyte (about 2000 blocks) of memory left on the disk.

Enhancements to the System Workspace

The following enhancements have been made to the System Workspace. See p. 503-509 of the Goldberg book for explanations of the other expressions in the System Workspace.

Create File System

Disk ← FileDirectory directoryNamed: '/public'.
>   Make the Smalltalk home directory an absolute path, in this case set it to the */public* directory.

Disk ← FileDirectory currentDirectory.
>   Make the home directory float to the directory in which the Smalltalk-80 system was invoked. This is the value for Disk in the default standard image. If, for example, you are in the */public* directory when you bring the standardImage up, Disk is set to */public* and all files are written to this directory unless a complete path name is explicitly specified. If you are in the */bin* directory when you bring up the standard image, the Disk is set to */bin*. Be careful. You may not have permission to write to */bin* and most of the system directories. Attempts to write to a protected directory are denied.

Dependents

ControlManager allUnscheduledDependentViews
do: [:aView | aView release].
>   Ask the ControlManager to release all views which are unscheduled but still in Object's dependency dictionary. These unscheduled views may result from user interrupts or incorrectly written code.

Globals

Smalltalk frills: false. "for slow machines"

The normal behavior (with frills set to true) is to restore previously occluded windows. Do not repaint occluded windows (just leave these areas gray) if frills is false.

Smalltalk saveSpace: true. "for limited memory machines"
> Forms for each window are normally saved so that the windows can be quickly restored. To conserve memory, set the saveSpace flag to true so that these forms are not saved. In that case, a window is repainted the slow way each time it becomes active.

Display

DisplayScreen displayExtent: 1024@1024.
> The normal size for the 4404 virtual screen is 1024 by 1024 pixels.

DisplayScreen displayExtent: 640@480.
> 640 by 480 pixels is the size of the visible screen. If this expression is executed, the only portion of the screen that is updated is the upper left viewport.

Display getViewportLocation
> Examine the origin (upper left corner) of the current viewport, that part of the virtual screen that is visible.

Display setInverseVideo
> Display white characters on a black background.

Display setNormalVideo
> Display black characters on a white background.

Measurements

MessageTally spyOn: [Behavior compileAll]
> Open an edit window on the performance analysis of the block expression.

MessageTally spyOn: [Behavior compileAll] to: 'spy.results'.
(FileStream oldFileNamed: 'spy.results') edit.
> Write the spy results to a file and open an edit window on its contents.

Miscellaneous Programming Tips

Debugging Smalltalk Code

When you are modifying methods or developing new ones, you might insert the following expressions into appropriate places in your code:

> self halt

> self halt: 'LabelString for the Notifier'

> Transcript show: 'counter=' , counter printString; cr

> (Delay forSeconds: 3) wait

> Cursor wait showWhile: [Sensor waitButton]

> Sensor leftShiftDown ifTrue: ["put some expressions here"]

You may create breakpoints by sending a message to an object to halt. Sometimes you may wish to print out debugging information to the System Transcript. If you're debugging something like displaying text in a window, halting or sending text to the System Transcript would cause an infinite loop. In these cases, you might wish to pause at appropriate places in the execution by delaying for a specified number of seconds or just waiting until a mouse button has been pressed. The last expression would allow you to try dangerous code or print extensive debugging information only part of the time, since the block expression is executed only when the left shift button is depressed.

Redefining =

Since "=" is such a fundamental operation, you should be very careful if you redefine it. You may also need to redefine hash at the same time (see Goldberg and Robson p. 96).

Uploading Files to Another Computer System

You may want to upload files from your Smalltalk-80 system to another computer system. The UNIX operating system has been chosen here for the procedure, but most of the steps apply to other operating systems. Follow this basic procedure:

1. If you wish to save text from a Smalltalk workspace, create a new file using a File List window, and copy the text from the workspace into the bottom pane. Choose the menu item *put* to write out the contents of the File List window into the new file. You might also *file out* code from a Browser.

2. Exit the Smalltalk-80 system with a System Menu *OS shell* choice. Establish an RS-232 connection with the Unix computer and the 4404 AI System by physically connecting an RS-232 cable and doing whatever you must (call up on a modem, use a "TIA" device, etc.) to communicate to the Unix system. Type the 4404 operating system command *remote* to establish communication between the 4404 OS and the RS-232 port.

3. Login to the Unix system. Use the xfer program with the *-gm* option. (Note that the xfer program is an unsupported 4404 program supplied as a C language source file that you must compile — and possibly edit to install — on your Unix system. You can transfer this xfer source file to your Unix system by:

   a. Typing *cat >xfer.c* on your Unix system.

   b. Using the F1 key to return to the 4404.

   c. Typing *list /samples/xfer.c >/dev/comm* on the 4404. (*xfer.c* is in */samples* as the 4404 is shipped from the factory.)

   You should now have a copy of *xfer.c* on your Unix system that you can compile and use to transfer files with error checking and carriage return to linefeed character translation.

Errors in the Addison-Wesley Books

The following is a list of errors in the Goldberg book (imprint 1984):

- On page 83, in Figure 5.5, the syntax diagram for "symbol" needs a path out analogous to "keyword".

- On page 88, in the first paragraph, ((sum count)) should be ((sum/count)).

- On page 126, the icons for saving and retrieving a FormEditor form source should be reversed.

The following is a list of the errors in the Goldberg and Robson book, which was reprinted with corrections, May 1983.

- On pages 127 and 129, it should refer to printStringRadix: instead of radix:.

- On page 202, the WriteStream instance protocol should list crtab and crtab: instead of crTab and crTab:.

- On page 289, it says to add the method to the *instance creation* protocol of class Class, but it should be added to the instance protocol named subclass creation.

- On pages 333 and 338, it refers to the class Bitmap. This class has been renamed WordArray.

- On page 399, it shows the wait cursor as three dots instead of the current hourglass shape. The message to Cursor for the crosshair is crossHair instead of crosshair.

Multiple Inheritance of Classes

The Smalltalk-80 system supports multiple inheritance of classes which allows objects to inherit methods from two or more super classes. Ordinary subclassing only allows inheritance from a single class.

For an example of how to use multiple inheritance, go to the System Browser and choose *Collections-Streams* and Stream. Then choose *hierarchy* from the middle mouse button. In the bottom pane, you see this hierarchy:

```
Stream ()
    PositionableStream ('collection' 'position' 'readLimit' )
        ReadStream ()
        WriteStream ( 'writeLimit' )
        ReadWriteStream ()

            ...
```

Instead of using this single inheritance scheme, ReadWriteStream could be a subclass of both ReadStream and WriteStream. Make the class NewReadWriteStream be a subclass of both by entering the following code in place of the normal class template for ReadWriteStream:

```
Class named: #NewReadWriteStream
    superclasses: 'ReadStream WriteStream'
    instanceVariableNames: "
    classVariableNames: "
    category: 'Collections-Streams'
```

When you *accept* this code, the system generates *conflicting inherited methods* messages in the System Transcript because, in the case of identical messages, it does not know which methods from ReadStream or WriteStream it should inherit. You can use the Browser to eliminate the sources of these errors by specifically indicating which of the conflicting methods to use or by removing the sources of conflict.

For more information, refer to the paper by Alan H. Borning and Daniel H. H. Ingalls, "Multiple Inheritance in Smalltalk-80" , pp. 234-237, *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982.

About Model-View-Controller

This section presents a brief conceptual overview to the topic of models, views, and controllers. An understanding of models, views, and controllers allows you to create Smalltalk applications that make sophisticated use of windows.

A model is the data that you want to represent on the Smalltalk screen. The model may be a string of text, an array of coordinates representing a waveform, a bitmap representing a digitized picture, etc. The model contains no information about how to display itself. This is the job of a view.

A view is the mechanism that a model uses to display itself in windows on the Smalltalk screen. A view knows how to display the information in a model. There are many kinds of specialized views such as the classes FormView, ListView, StandardSystemView, and BrowserView. These views are suited to displaying the special kinds of models their names suggest.

A controller is the mechanism that you use to interact with a model displayed in a window. A controller also interacts with a controller manager so that activity in multiple windows or multiple views within one window is appropriately coordinated. A controller pays attention to cursor position and mouse button activity.

The Model-View-Controller Triad

Controllers and views are linked internally by instance variables that refer to each other and to the model. But the model does not refer directly back to its own controller and view. The idea is to keep models simple by not requiring them to know how to display themselves.

So, in order to display information on the Smalltalk screen, you need to deal with the mode-view-controller triad. Roughly speaking, each single-paned window has one model-view-controller triad associated with it. And, by extension, each multi-paned window, such as the System Browser, has a model-view-controller triad for each pane. Furthermore, each subview for each pane is eventually related hierarchically to a top view that controls the entire set of subviews for the whole window. See Figure 6 and Figure 7.

View Displaying Protocol

When a view is asked to display itself by some update action received by the controller, as when you choose the *restore display* command from the System Menu, each view goes through this sequence of messages:

    displayBorder
    displayView
    displaySubViews

Each kind of view knows how to display the special kind of model information that it is suited to. You can use the System Browser to explore how different kinds of views display themselves.

5606-6
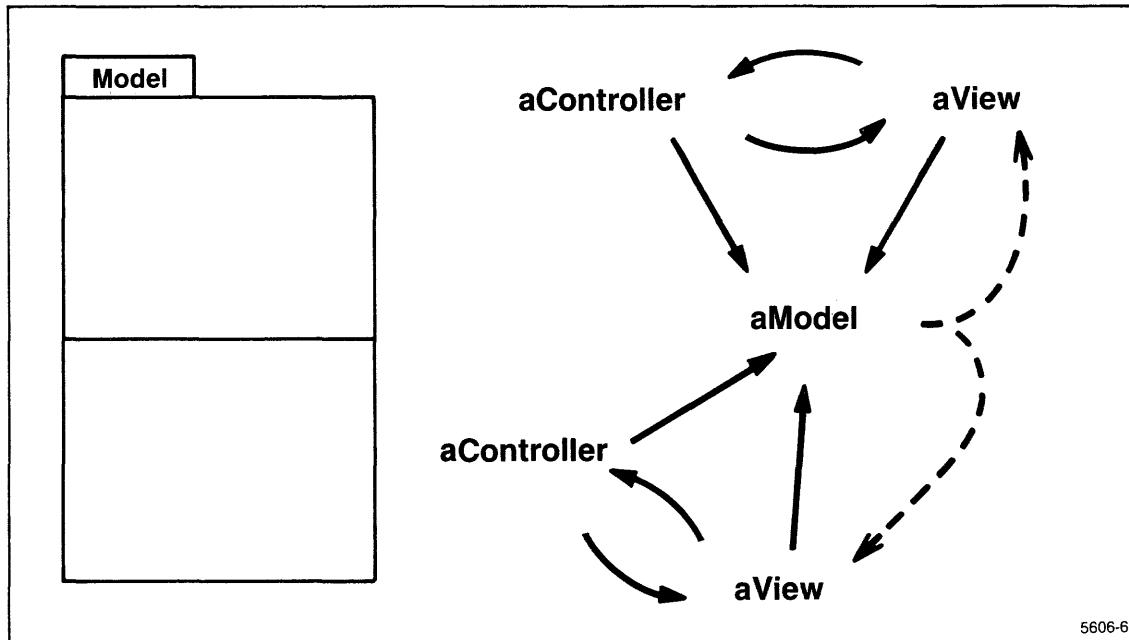
Figure 6. Double Window With Two Model-View-Controllers.

How to Construct a Window

Here is some Smalltalk-80 language "pseudocode" to give you an idea of how to construct a side-by-side, two-paned window.

```
leftSubView ← NewKindOfView new
    model: aNewKindOfView;
    borderwidth: 2;
    insideColor: Form white.

rightSubView ← NewKindOfView new
    model: aNewKindOfView;
    borderWidth left: 0 right: 2 top: 2 bottom: 2;
    insideColor: Form white.

topView ← StandardSystemView new
    label: 'NewKindOfView Window';
    addSubView: leftSubView;
    addSubView: rightSubView toRightOf: leftSubView.

topView controller open
```

The pseudocode assumes, of course, that you have already created NewKindOfView by creating a subclass of the class View or one of the many kinds of views that are part of the standard system, such as ListView, WorkspaceView, and so forth. Note that the pseudocode above representing the instance creation of the window must take care of arranging the subviews properly in the top view. The first three blocks of pseudocode construct the window, and the final line of code tells the topView's controller to schedule this window for display and make it the active window.
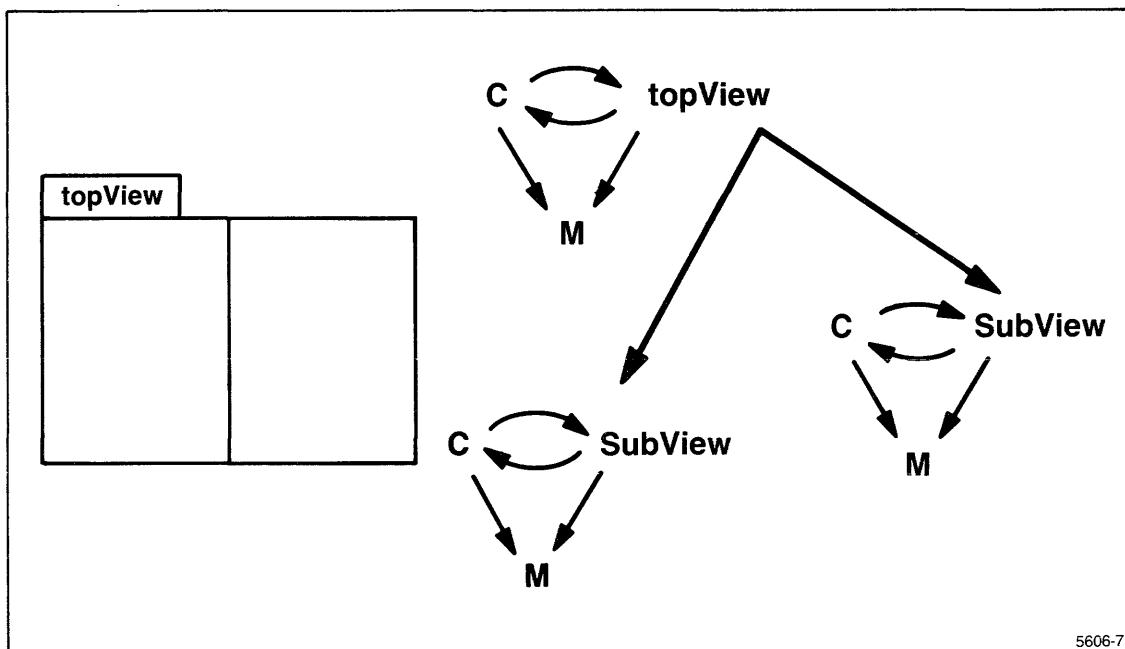

Figure 7. Hierarchy of Model-View-Controllers.

Controllers and Controller Protocol

If you look at the class Controller, you see that, like the class View, it has quite a few subclasses which are specialized controllers. Some of these are MouseMenuController, ScrollController, ListController, StandardSystemController, and so forth. You may decide that none of these controllers fits your application and create your own, perhaps, as a modification of MouseMenuController.

Each controller receives this basic sequence of messages:

```
controlInitialize
controlLoop
   isControlActive
   controlActivity
controlTerminate
```

The overall controller manager of the system, the global variable ScheduledControllers, constantly monitors the position and activity of the mouse. See Figure 8. When mouse button activity occurs within a window, the controller manager usually passes control to that window's controller, and the window's controller then executes the sequence shown above. An important point to note is that each individual controller only gives up control when it wants to. The controller manager cannot take control away from an individual view controller. (You can modify this if you want for your own applications, though.) After an individual controller gives up control, the controller manager continues to poll all of the currently scheduled controllers until it finds that another one wants control.

A Conceptual Example

Suppose you wanted to interactively allow a user to change the routing of wires on a prototype wire wrapped circuit board. The model would be an ordered array of numbers in some coordinate system related to the dimensions of the circuit board. Coordinate pairs would represent the positions of endpoints of wires. And the ordered array would then represent a wire list. See Figure 9.

To realize this in the Smalltalk-80 system, you would have to create an appropriate view and controller for the wire list model data. You might come up with something like the following Smalltalk "pseudocode".

You would have to define the method display for the class WireListView so that it would know how to display itself. This is one way to do that:

```
I wire I

wire ← LinearFit new.
model do:
    [:each I wire add: each].
wire display
```

The class LinearFit is a subclass of DisplayObject and Path so it has quite a bit of intelligence that you can make use of.

You would also need to define a method selector controlActivity to let the class WireListController know what to do. This is one way to do that:

```
I down up I

Sensor redButtonPressed
    ifTrue:
    [down ← Sensor cursorPoint.
     up ← Sensor waitNoButton.
     model
        replace: (model pinNear: down)
        with: up ]
```
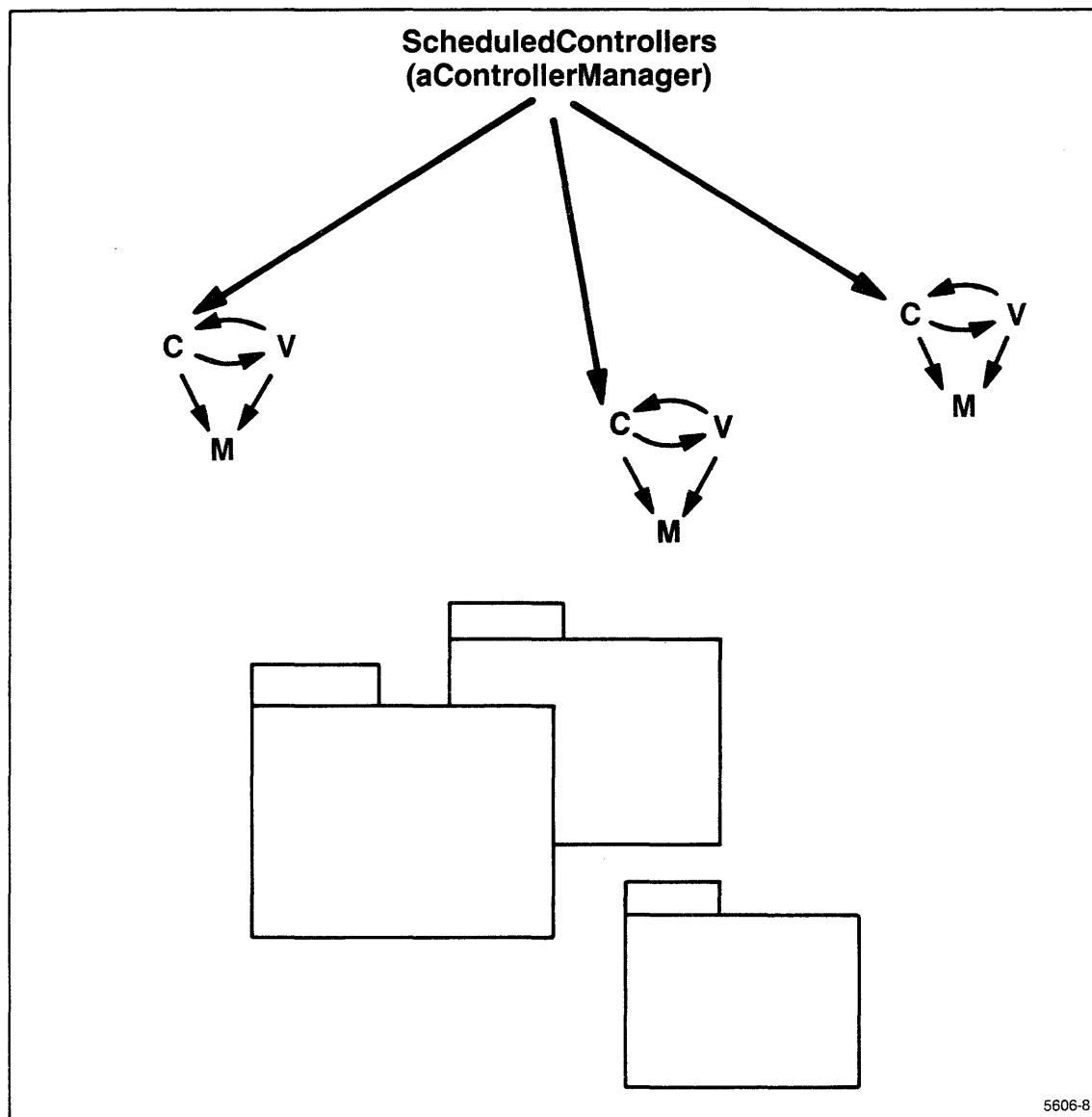
Figure 8. ScheduledControllers for Three Model-View-Controllers.

This code assumes that the user is looking at a window with the wire list endpoints marked and that the user wants to select a marked endpoint and move it to another location. The message selector pinNear: is a method that you have to write that decides what endpoint the user intended to select.

Viewports and Windows

You may have realized as you went through this conceptual example that the screen coordinates of the Smalltalk-80 system and the coordinates of the wire list are probably not the same. Look up the class WindowingTransformation in the System Browser. In the definition of the class, you find the instance variables window and viewport used. The variable window is a Rectangle in the coordinate system of the model (the wire list coordinates). The variable viewport is a Rectangle in the "destination" coordinate system. In this example, the viewport coordinate system is the screen coordinate system − 1024 by 1024 pixels.

A view's window and viewport are used to form a WindowingTransformation. See Figure 10. To make use of this transformation, you would modify the previous example view to include:

```
model do:
    [:each I wire add: (self displayTransform: each)]
```

Alternatively, the sample controller can perform the inverse transformations with:

```
down ← (view inverseDisplayTransform: Sensor cursorPoint)
```

In a window with subviews, two transformations would be required. See Figure 11. The first transformation would carry coordinates from the subview window coordinate system to the subview viewport coordinate system. The subview viewport coordinate system would actually be the same coordinate system as the top view's window coordinate system. The second transformation would then carry coordinates from the top view's window coordinate system (which is equal to the subview's viewport coordinate system) to the top view's viewport coordinate system, which is, finally, the 1024 by 1024 screen pixel coordinate system.

This brief overview of models, views, and controllers is meant to get you started. For more information, turn to the System Browser and examine the class View and its specialized view subclasses, the class Controller and its specialized controller subclasses, and the class WindowingTransformation.

Wire List Diagram
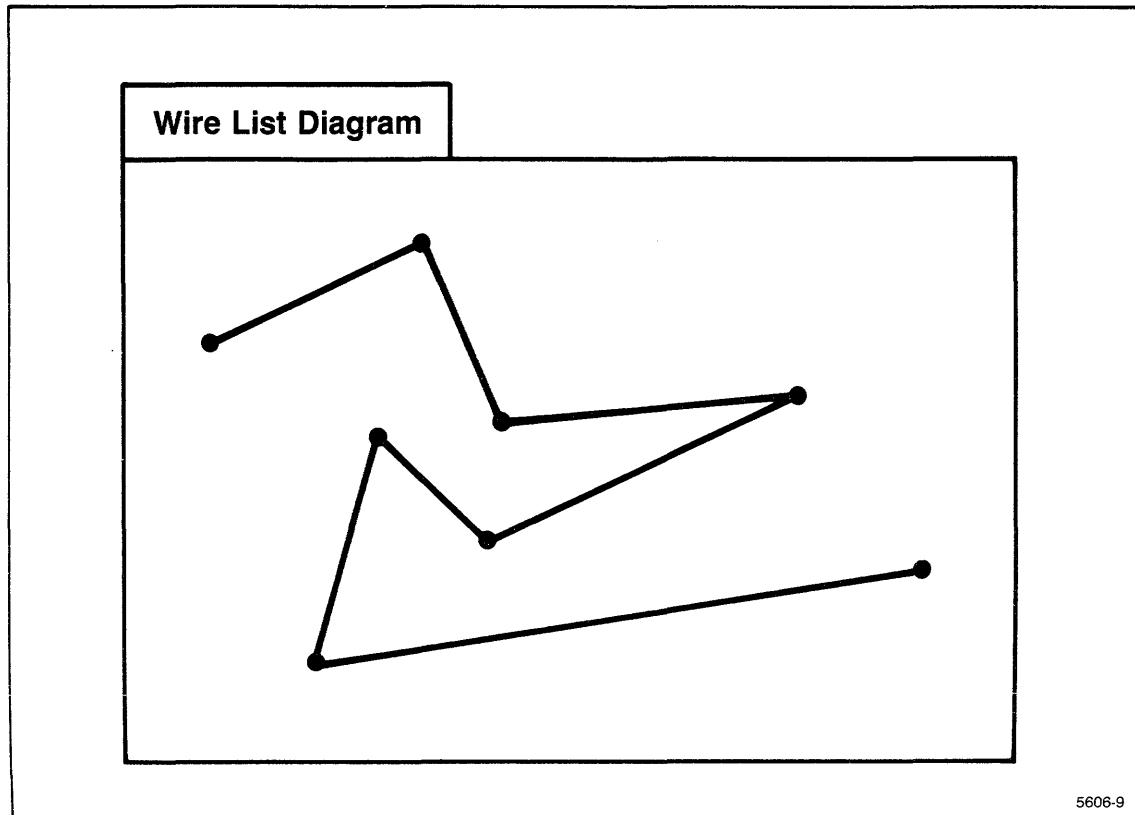
Figure 9. User View of Wire List Example.
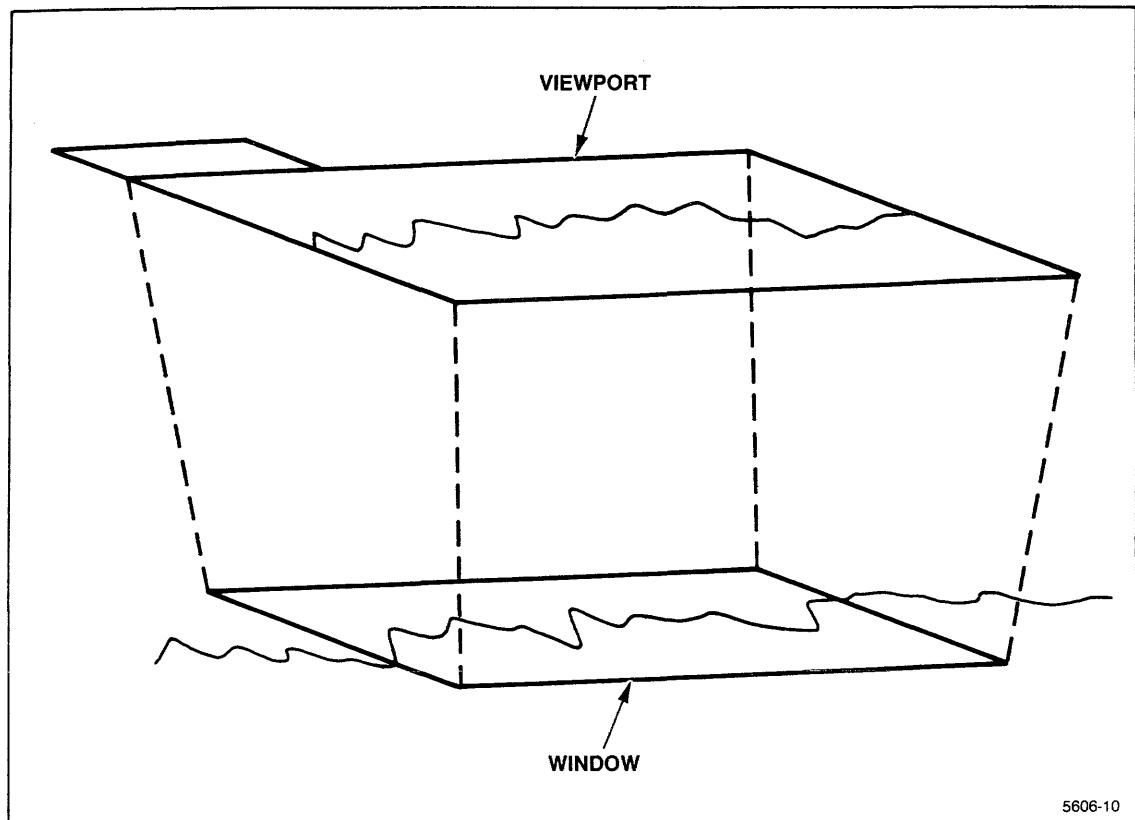
VIEWPORT

WINDOW

Figure 10. Window/Viewport Relationship.

SubView
displayBox

topView viewport

topView window

subView viewport

subView window

5606-11
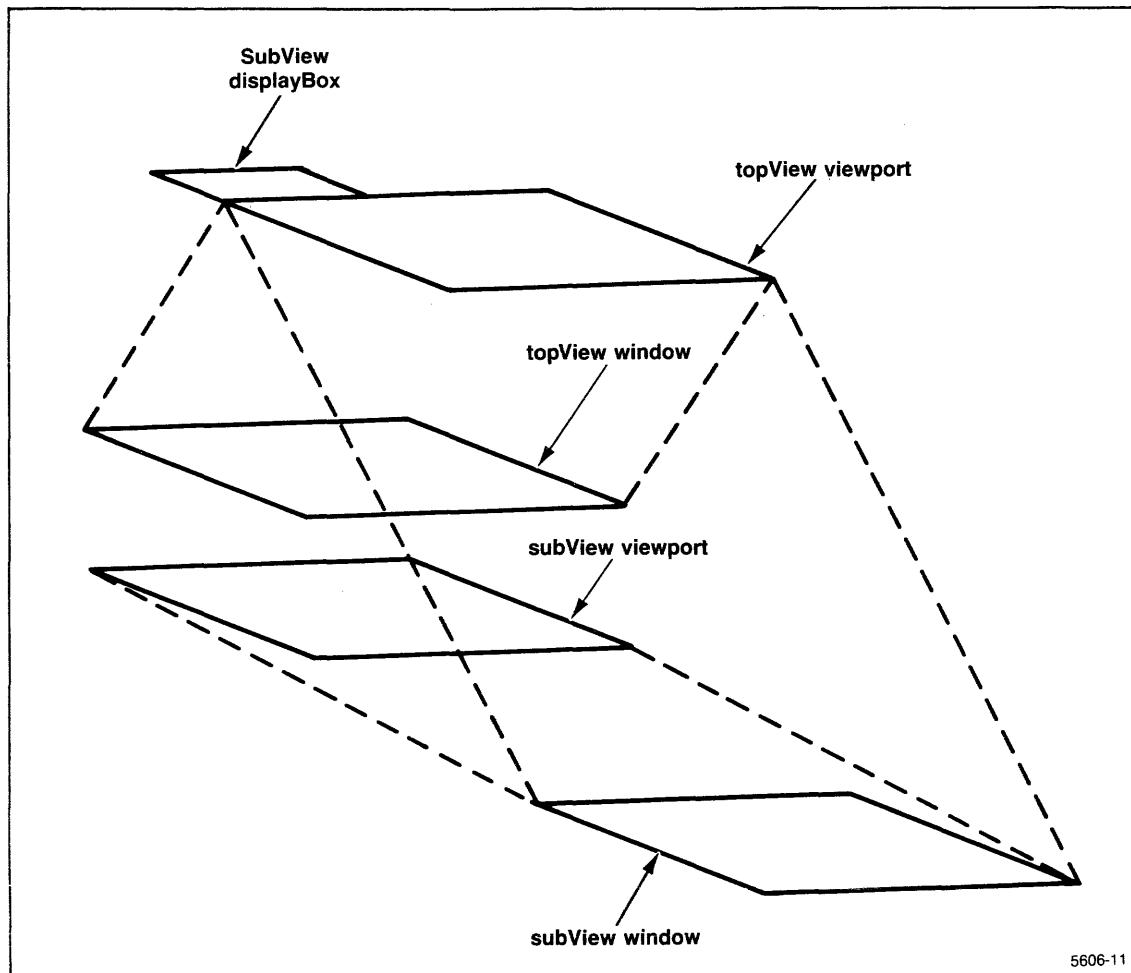
Figure 11. Composition of Multiple Window/Viewport Transformation.

SECTION 5

4404 SMALLTALK REFERENCE

The information in this section primarily documents additions and enhancements that have been made to the Smalltalk-80 System Version 2 in order to create the 4404 version of the Smalltalk-80 system. Most of the additions and enhancements involve the interface between the 4404 operating system and the Smalltalk-80 system. The current release of the 4404 Smalltalk-80 system is Version T2.1.

The Smalltalk-80 System in the 4404 OS Environment

Making System Calls

Sometimes you may want to make 4404 operating system calls from the Smalltalk-80 system. There are a number of methods supporting system calls in the 4404 Smalltalk-80 system. These methods are defined in the class TekSystemCall. You can use the System Browser to find these methods. Choose category *Files-Support*, class TekSystemCall, and message categories: *display instance creation* or *system dependent operations* to see the system call methods. The message category *display instance creation* deals with display operations. The message category *system dependent operations* deals with a low level interface to the operating system. Refer to the 4404 AIS Reference Manual for a complete description of system calls.

The message selectors found in the *display instance creation* category all have higher level methods that you can access to achieve the same functionality, so you will probably not use these system calls. Go to the System Browser and select these: category *Graphics-Display Objects*, DisplayScreen, and message category *display functions*. For example, choose setInverseVideo. You can see that the method uses the TekSystemCall method whiteOnBlack in a simple and straightforward manner. This is true of all of the *display instance creation* TekSystemCall methods.

Two System Call Examples

To see how to use the system call methods, look at two examples from the system itself. Use the System Browser to find the senders of open:mode: and link:to:. Click on the following selections: category *Files-Support*, class TekSystemCall, message category *system dependent operations*, method link:to:. Find all the senders of link:to: by choosing the

middle button *senders* command. You will see a *senders of link:to:* window open. Choose *TekSystemCall class rename:as:*. Note that the method rename:as: uses the system call method link:to: in the second line of code. self is understood as TekSystemCall in this line. The link:to: method creates a data structure for making a system call, and the method value causes the link system call to be executed and detects errors.

```
rename: aFileName as: newFileName
        "Rename the file named aFileName to have
        the name newFileName.  Create an error if
        aFileName does not exist; but not if
        newFileName exists."

        (self unlink: newFileName) valueIfError: [].
        (self link: aFileName to: newFileName) value.
        (self unlink: aFileName) value.
```

Find the next example by selecting open:mode: in the message selectors pane of the System Browsers and opening a *senders of open:mode:* window. Choose *TekSystemCall class open:* to see the following code.

```
open: aString
        "Open the file named aString.   Answer a readWrite
        fileDescriptor for the file."

        | sysCall |
        sysCall ← self open: aString mode: 2.
        sysCall value.
        ↑sysCall DOout
```

(Note that the the ASCII up-arrow is the return arrow in Smalltalk code.) aString is a file name and mode 2 means to open the file for both reading and writing. These arguments correspond to the arguments described in the 4404 AIS Reference Manual in the Section 6, System Calls. Look up the system call named *open*. Note also that this method returns the file descriptor in register D0, corresponding to the documented *open* system call interface.

List of System Calls

The following is a list of operating system calls accessible from the Smalltalk-80 system. You can find these in the System Browser by choosing the *Files-Support* category, TekSystemCall class, and *system dependent operations*. Thus, these are all methods that are understood by the class TekSystemCall. Note that nearly all the method names are taken from the names of system calls as they are documented in the 4404 AIS Reference Manual.

```
chacc: fileName mode: permissions
```
        Checks the accessibility of a file with respect to specified permissions.

```
chdir: directoryName
```
        Changes directory to a new directory name.

```
chown: fileName to: ownerID
```

Changes the owner of a fileName to ownerID.

chprm: fileName to: permissions
Changes the permissions for a file.

close: fileDescriptor
Closes a file.

create: fileName mode: modeBits
Creates a new file.

crPipe: fileDescriptor
Creates a pipe.

crtsd: newName mode: mode addr: addr
Creates a directory or special file.

defacc: permissions
Sets the default permissions as specified by the operating system command *perm*.

dup: fileDescriptor
Duplicates the file descriptor; opens the file again.

dups: fileDescriptor with: specifiedDescriptor
Duplicates the file descriptor, specifying the file descriptor of the duplicated open file.

filtime: fileName to: time
Sets the last modified time of a file.

fork
Creates a new task.

getId
Gets the running task's ID.

getuId
Gets the actual user ID and the effective user ID.

link: fileName to: linkName
Create a link to a file.

lrec: fileDescriptor howmany: count
Makes an entry in system's locked record table.

ofstat: fileDescriptor buffer: buff
Gets the status of an open file.

open: fileName mode: modeBits
Opens a file.

read: fileDescriptor buffer: buff nbytes: numberOfBytes
Performs a read operation into the buffer.

seek: fileDescriptor offset: position whence: start
Positions a file's read/write pointer relative to start.

setpr: priority

Sets the priority.

setuid: userID

Sets the actual user ID and the effective user ID to userID.

spint: taskNumber an: interrupt

Sends a program an interrupt.

status: fileName buffer: buf

Reads the status information of a file into the buffer buf.

term: terminatingStatus

Terminates a task with the status terminatingStatus where a zero status means no error occurred.

time: tbuff

Returns the system's current time in the buffer tbuff.

truncate: fileDescriptor

Truncates a file at the current position.

ttime: tbuff

Returns accounting time information about a task in the buffer tbuff.

ttyget: fileDescriptor buffer: ttybuff

Ttybuff should be a WordArray of size 3.

ttynumber

Gets the number of the calling task's terminal.

ttyset:fileDescriptor buffer: ttybuff

Sets the tty information as described in ttyget.

unlink: fileName

Unlinks a file.

update

Update the information on the disks.

urec: fileDescriptor

Unlocks a file record.

wait

Waits for a child or a program interrupt.

write: fileDescriptor buffer: buff nbytes: numberOfBytes

Writes the contents of the buffer to a file.

4404 Primitive Methods

All primitive methods described in the Goldberg and Robson book in Chapter 29 are implemented in the 4404 Smalltalk-80 system. These include all the optional standard primitive methods. In addition, the 4404 Smalltalk interpreter implements a number of implementation-dependent primitives. They are listed here along with a brief description of each one.

Primitive #129 namedSnapshot:

This is a primitive method in class SystemDictionary. Its argument is a file name. Invocation of this primitive causes a snapshot of the currently executing virtual image to be written to the file. If the call is successful, the primitive returns nil or self, otherwise it fails. If the image was just written, the returned value is nil. If it is being restored, the returned value is self.

Primitive #130 primitiveIncrementSP

This is a primitive method in class ContextPart. It adds one to the stack pointer field of the receiving context and stores nil into the new top of stack element. This primitive and primitive #131 are the only acceptable ways to explicitly modify a context's stack pointer.

Primitive #131 primitiveDecrementSP

This is a primitive method in class ContextPart. It subtracts one from the stack pointer field of the receiving context. This primitive and primitive #130 are the only acceptable ways to explicitly modify a context's stack pointer.

Primitive #134 primSysCall

This is a primitive method in class TekSystemCall. It causes the operating system call specified by the receiving object to be executed. If the system call is executed without error, true is returned. If an error occurs during the system call, then false is returned.

Primitive #135 primDispCall

This is a primitive in class TekSystemCall. It specifies a display-related system call to the operating system. If the call is executed without error, true is returned. If an error occurs during the call, then false is returned.

Primitive #136 primShell

This is a primitive method in class ScreenController. It forks a shell from the operating system with a duplicate of the parent shell's history. When the shell exits, control is returned to the Smalltalk-80 system. true is returned unless the system call fails, in which case false is returned.

4404 Smalltalk-80 Virtual Image Enhancements
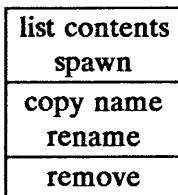
A number of image enhancements have been made to the Smalltalk-80 Version 2 virtual image.

File Lists and Directory Browsers

The Files and File List Browsers are described in the Goldberg book in Chapter 22. The File System has been extended to support the creation and browsing of directories. The wildcard characters "*" and "#" can only be used in the file name itself and not in the path name. The asterisk matches any number of characters and the pound sign matches a single character. For example, /public/st* is acceptable, but /pu*/junkfile is not acceptable.
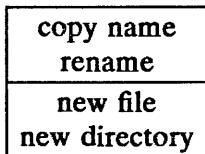
If a name is selected in the second pane of a FileList, a comment is displayed in the bottom pane. The size and last modification date and time is displayed for existing files. For directories, a message states that the selected file is a directory. If a new name is accepted in the first pane of a FileList, the comment -new file or directory- is displayed.

If the selected name is a file, the regular middle button menu is used in the second pane. The first two items in this menu (get contents and file in) are not applicable for directories. Thus, if the selected name is a directory, the menu is:

list contents
spawn
copy name
rename
remove

The *list contents* selection lists the names of files within the directory in the bottom pane. The *spawn* selection opens a new FileList on all the files within the directory. This is similar to selecting *spawn* in one of the panes of a regular Browser. The *remove* option removes the directory if it is empty. If it is not, a notifier appears saying that the directory cannot be removed because it is not empty.

If a new name is accepted in the first pane of a FileList and that name is selected in the second pane, the middle button menu is:

copy name
rename
new file
new directory

The *copy name* and *rename* options function as usual. The *new file* option creates an empty file with the specified name and replaces the middle button menu with the normal middle button menu for files. If you select the *new directory* option, the specified name becomes a directory and the middle button menu is switched to the directory menu.

When the *get contents* command is selected on a regular file, the size is checked. If a file is too large to fit in a String, that is, its size is greater than 65535 bytes, a notifier appears with the message "file too large to edit".

Printer Support

The *copy display* command in the System Menu prompts for a file name and then copies the screen bitmap to that file. This file is written in the same format as that of a Form. The file can then be sent to the printer from the operating system. To print a bitmap from a file called *screen.bm* on a Tektronix 4644 printer, type:

/samples/printer/bprint screen.bm

The *bprint* command defaults to double density, which means that the print out looks half as wide as it should in relation to its height. If you use the +*s* option on the *bprint* command, the aspect ratio looks much more like the real screen, but the last few pixels on the right side of the screen are missing.

The *print out* menu option in the System Browser writes the code out to a file with a *.pp* extension. This code cannot be directly filed into a Smalltalk-80 image, but is intended to provide a more human readable format than that provided by the *file out* menu selection. It can be regarded as a pretty printed version, although no automatic formatting is performed on methods.

The *.pp* files contain control characters for bold, italic, and normal fonts taken from from ANSI Standard X3.64.

Note that changes that you file out from a Browser are removed from the System Change Set. Refer to the Goldberg book for more about this. Entries that you print out from a Browser are not removed.

From the operating system, you can look at a file named *Collection.pp* by typing:

list Collection.pp

(There are no italic fonts for the terminal, so italic entries are printed in the normal font.)

To print the file on a Tektronix 4644 printer, type:

/samples/printer/print +s Collection.pp

An example of this output might look like:

```
Object subclass: #Collection
       instanceVariableNames: "
       classVariableNames: "
       poolDictionaries: "
       category: 'Collections-Abstract'
```

Collection comment: 'I am the abstract class of all collection classes.'

Collection methodsFor: 'accessing'

size
```
       "Answer how many elements the receiver contains."

       | tally |
       tally ← 0.
       self do: [:each | tally ← tally + 1].
       ↑tally
```

Collection methodsFor: 'testing'

Includes: anObject
```
       "Answer whether anObject is one of the receiver's elements."

       self do: [:each | anObject = each ifTrue: [↑true]].
       ↑false
```

IsEmpty

"Answer whether the receiver contains any elements."

↑self size = 0
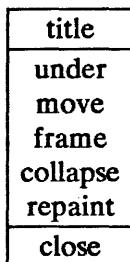
occurrencesOf: anObject
"Answer how many of the receiver's elements are equal to anObject."

I tally I
tally ← 0.
self do: [:each I anObject = each ifTrue: [tally ← tally + 1]].
↑tally

Title Changes

The title menu option is an addition to the right button menu. When it is selected, you are asked for the name of the new title for the window. The title tab is then repainted with the new name.

title
under
move
frame
collapse
repaint
close

Window Management

Windows do not unnecessarily repaint themselves. When a window is moved or closed, the parts of the visible windows immediately underneath it are repainted. The *repaint* menu item in the right button menu causes a window to be repainted to clean up from unscheduled graphics.

By default, each time a window is de-emphasized, the visible contents of the window are saved in a form. This form allows the window to be displayed quickly the next time it becomes active. (The System Transcript window is an exception to this convention since it can be modified while inactive.) Since each window has a corresponding form, the existence of many windows can use a considerable amount of memory. An option is provided to turn off the default storage of these forms. To exercise this option, execute the expression Smalltalk saveSpace: true.

In conjunction with the saving of forms for windows is the redisplay of parts of inactive windows which have been obscured. For instance, if one window were covering a second window and the first window is closed, then the part of the second window which was previously covered by the first window would be redisplayed. The saved form for the first window is used to redisplay the obscured portion. (In the case of redisplay of the System Transcript window, the obscured portions are colored white). The existence of many obscured windows needing to be redisplayed can be time consuming. An option is provided to turn off the default redisplay of obscured windows. To exercise this option, execute the expression Smalltalk frills: false.

Several combinations of these two options exist. One may wish to save the window forms in order to display windows quickly, but not want to redisplay obscured windows. In this case, set both saveSpace and frills to false.

Another reasonable combination is to redisplay obscured windows but not save forms. Since the forms associated with windows are not saved, obscured parts of windows are filled in with a white form. This combination may be used by people who want to limit memory usage and yet want to know explicitly where all the windows are. In this case both saveSpace and frills would be true.

Cursor Center Key

The cursor center key (F12) moves the cursor to the center of the visible viewport. If the cursor is unlinked when the key is pressed, it is re-linked and the normal cursor is displayed. A portion of the cursor form is normally constrained to be within the bounds of the virtual screen. But, depending upon the shape of the cursor within its 16 by 16 pixel cursor Form, you may not always be able to see it.

File System Interface

To open a previously existing file for reading or writing, send the message oldFile: with the file name as an argument to an instance of a FileDirectory such as:

    f ← Disk oldFile: 'timingData'

Executing this expression insures that the variable f is an instance of a FileStream on an existing file. A notifier appears if the file does not exist.

To open a new file, send the message newFile: to an instance of a FileDirectory as in:

    f ← Disk newFile: 'testCases'.

This guarantees that f is a FileStream on a new file named *testCases*. If a file in the Disk directory previously exists with that name, a confirmer tells you that the file already exists. If you *proceed*, the existing *testCases* file is renamed as 'testCases.bak' and a new *testCases* file is created.

The file: message delays the opening of a file until a data transfer takes place.

    f ← Disk file: 'testResults'

If such an instance of a FileStream is sent the nextPut: message, and a file with that name already exists, the file is automatically backed up with a *.bak* extension and the data is written to the new file. If the file *testResults* does not exist and a read operation is attempted, an error is produced. If the file *testResults* does not exist and a write operation is attempted, the file is created, opened for reading and writing.

One can query the existence of such a file before doing input or output. For example:

    f ← Disk file: 'testResults'.
    f exists

answers true if the yet-to-be opened file already exists.

Sending the message readWrite to a FileStream created with the file: message insures that no backup is made. This allows updates in place.

For special devices like the sound device, /dev/sound, which can only be opened for writing, use the oldWriteOnlyFile: message.

```
f ← Disk oldWriteOnlyFile: 'dev/sound'.
f binary.
f nextPut: soundDevByteArrayBuffer.
f close.
```

The oldWriteOnlyFile: message says that this file is intended to be a previously existing file (hence, do not employ the back-up mechanism) which is to be opened strictly in a writeOnly mode.

The file system is sophisticated enough to manage the opening and closing of files fairly transparently. An attempt to read a closed file causes the file to be opened. If an attempt is made to open a file that is referenced by another open FileStream, an attempt is made to close the already open file. Its position is retained so that later data transfers can resume unaffected by the closure.

It is still a good idea to close files explicitly. Closing forces all buffers to be flushed to the physical file, thus protecting from unexpected data loss in the event of a crash.

Framing Rectangular Regions

The standard frame specification method reverses the rectangular region.

Workspace Variables

If a previously undefined variable is assigned a value within a workspace through an assignment statement, the variable is automatically declared as a local variable. Its scope is defined for the entire duration of the workspace or until it is assigned the value nil.

The Smalltalk compiler defines and later references variables in the context of invoking workspaces. The spelling corrector is not invoked when an undefined name appears on the left of an assignment. The Smalltalk debugger, without reference to the original workspace, is able to correctly recompile a method's source code. Since each workspace has its own dictionary of local variables, objects that are referenced by a workspace variable exist for the lifetime of the workspace. If you are concerned with memory space usage, assign nil to all the variables within a workspace or copy the text from one workspace into a new one and delete the old workspace. Within each workspace, you can examine the current contents of the workspace variable dictionary by evaluating the expression WorkspaceVariables inspect or by asking the object named WorkspaceVariables to print itself.

4404 and Smalltalk Version 2 User Interface Differences

Although the 4404 version of the Smalltalk-80 system and the Goldberg book version of the Smalltalk-80 system are very similar, there are some differences. A list of these differences follows:

On page 6, Chapter 1: The 4404 virtual screen coordinates are 1023,1023 at the lower right corner. The visible part of the screen is 640 pixels wide by 480 pixels high. The joydisk on the keyboard does a hardware scroll through the entire range of virtual screen

coordinates.

On page 7, Chapter 1: The 4404 keyboard has a rubout key (labeled *Rub Out*) instead of a delete key. The backspace key is labeled *Back Space* and the line feed key is labeled *Line Feed*. There is also a *Caps Lock* key. The 4404 keyboard also has special keys for the assignment arrow and the return arrow. The assignment arrow is the unshifted leftmost key in the upper row of keys. The return arrow is the shifted version of the same key. Also, the programmable key (F12) puts the arrow cursor in the center of the screen.

On page 14, Section 1.2, Getting Started: See the accompanying "Invoking the Interpreter" heading and Section 4 for how to get started.

On page 14, System Files: The 4404 Smalltalk-80 system directory is described under the "Smalltalk File Structure" heading in this section.

On page 24-26, Saving Information: When a snapshot is made, a prompter asks you for the name of the snapshot file. It is stored as that name, not *snapshot.im*. No command files are used.

On page 30-31, Section 2.1, Cursors: In addition to the cursors shown there, the 4404 Smalltalk-80 system has two more: a major and minor compaction cursor. These two cursors show up automatically when the Smalltalk-80 system is doing garbage collection-related chores. The minor compaction cursor has a leftward point arrow as part of its image. The major compaction cursor is a box with a slanted line in it.

On page 43, Section 2.4, The System Menu: The System Menu also has a hardcopy option, *copy display*, and a command that spawns a new shell in the 4404 operating system, *OS shell*. *copy display* copies the screen bitmap into a file that you specify, or you can choose the default file name with a carriage return. The entire bitmap of 1024 by 1024 pixels is copied to a file. The file is relatively large so make sure you have enough room on your disk. *OS shell* spawns a new shell in the 4404 operating system. Thus, you can "set aside" the Smalltalk-80 system while you do some things in the operating system. When you are done, return to the Smalltalk-80 system by typing the OS command *exit*.

On page 62: In the list of editing command keys, substitute rubout for the delete key.

On page 179: The Browser's *move* menu item now prompts with the name of the currently selected class and protocol.

On page 209-210: The 4404 file system has been simplified, which resulted in the elimination of the classes File and FilePage.

On page 354, Chapter 16, Spelling correction: If a previously undefined variable appears to the left of an assignment operator within a workspace, the variable is automatically declared to be a local variable when the expression is evaluated. Its scope remains for the lifetime of the workspace, or until it is assigned the value nil.

On page 435, Chapter 22: Print out sends the pretty-printed code to a file.

On page 440, File List Browser: Support for directories is added to FileLists: see the "File Lists and Directory Browsers" section of this document.

On page 460-461, Saving Your System State: The default snapshot name is not the same as the first part of the name found as the first element of the global array, SourceFiles. The extension *.im* is not appended to 4404 image names. A new changes file is not made

for each image; the name of the changes file remains unchanged.

On page 467, The System Audit Trail: When a snapshot has completed correctly, the changes file is marked with the date, time, and image file name in addition to the "—SNAPSHOT—" notation.

Smalltalk File Structure

The /smalltalk Directory

The */smalltalk* directory contains the *standardImage* file and the *system, demo,* and *fileIn* directories. The standardImage file is the default virtual image containing Tektronix enhancements. The interpreter is called */bin/smalltalk*. For more information about the contents of the sources, image, and changes files, see the Goldberg book, pages 457-458.

The /smalltalk/system Directory

The */smalltalk/system* directory contains the following system files:

- *standardSources* This contains the standard source code for each of the compiled methods in the standard image.

- *standardChanges* Initially empty, this file is the default changes file corresponding to the *standardImage*. This file may grow each time the *standardImage* is used.

- *initialization* This directory contains files that are needed for class initializations.

Additional source code

The */smalltalk/fileIn* directory contains additional source methods which may be filed into the image. A *README* file documents this code. This code is not supported and is distributed on an "as is" basis.

Demo material

The */smalltalk/demo* directory contains an image, changes file, and other source files which may be used for demonstration purposes. The code in these files is not necessarily fully debugged and is not supported. You can delete these files to save disk space.

- *demoImage*: This is an image with interesting applications not considered essential for the *standardImage*.

- *demoChanges*: The changes file for the demoImage.

APPENDIX A

SMALLTALK CLASSES LIST

This hierarchical list of all the classes in the 4404 version of the Smalltalk-80 system appears here for your convenience. You can generate the same list by selecting the category *Kernel-Object* and the class Object in the System Browser, and then by selecting *hierarchy* in the pop-up menu.

Object ()
  Behavior ('superclass' 'methodDict' 'format' 'subclasses')
    ClassDescription ('instanceVariables' 'organization')
      Class ('name' 'classPool' 'sharedPools')
        ... all the Metaclasses ...
        Metaclass ('thisClass')
        MetaclassForMultipleInheritance ('otherSuperclasses')
  Benchmark ('dummy' 'verboseTranscript' 'reporting' 'reportStream'
  'fromList')
  BinaryChoice ('trueAction' 'falseAction' 'actionTaken')
  BitBlt ('destForm' 'sourceForm' 'halftoneForm' 'combinationRule'
  'destX' 'destY' 'width' 'height' 'sourceX' 'sourceY' 'clipX'
  'clipY' 'clipWidth' 'clipHeight')
    CharacterScanner ('lastIndex' 'xTable' 'stopConditions' 'text'
    'textStyle' 'leftMargin' 'rightMargin' 'font' 'line'
    'runStopIndex' 'spaceCount' 'spaceWidth' 'outputMedium')
      CharacterBlockScanner ('characterPoint' 'characterIndex'
      'lastCharacter' 'lastCharacterExtent' 'lastSpaceOrTabExtent'
      'nextLeftMargin')
      CompositionScanner ('spaceX' 'spaceIndex')
      DisplayScanner ('lineY' 'runX')
    Pen ('frame' 'location' 'direction' 'penDown')
  Boolean ()
    False ()
    True ()
  Browser ('organization' 'category' 'className' 'meta'
  'protocol' 'selector' 'textMode')
    Debugger ('context' 'receiverInspector' 'contextInspector'
    'shortStack' 'sourceMap' 'sourceCode' 'processHandle')
    MethodListBrowser ('methodList' 'methodName')
  Change ('file' 'position')
    ClassRelatedChange ('className')
      ClassChange ()
        ClassDefinitionChange ('superclassName' 'classType'
        'otherParameters')
        ClassOtherChange ('type')
          ClassCommentChange ()
      MethodChange ('selector' 'category')
        MethodDefinitionChange ()
        MethodOtherChange ('type')
    OtherChange ('text')
  ChangeSet ('classChanges' 'methodChanges' 'classRemoves'
  'reorganizeSystem' 'specialDoIts')
  Checker ()
  ClassCategoryReader ('class' 'category')
  ClassOrganizer ('globalComment' 'categoryArray' 'categoryStops'
  'elementArray')
    SystemOrganizer ()
  Collection ()
    Bag ('contents')

MappedCollection ('domain' 'map')
SequenceableCollection ()
  ArrayedCollection ()
    Array ()
    ByteArray ()
      CompiledMethod ()
    RunArray ('runs' 'values')
    String ()
      Symbol ()
    Text ('string' 'runs')
    WordArray ()
      DisplayBitmap ()
    Interval ('start' 'stop' 'step')
      TextLineInterval ('internalSpaces' 'paddingWidth')
    LinkedList ('firstLink' 'lastLink')
      Semaphore ('excessSignals')
    OrderedCollection ('firstIndex' 'lastIndex')
      SortedCollection ('sortBlock')
  Set ('tally')
    Dictionary ()
      IdentityDictionary ('valueArray')
        MethodDictionary ()
      LiteralDictionary ()
      SystemDictionary ()
    IdentitySet ()
Compiler ('sourceStream' 'requestor' 'class' 'context')
Controller ('model' 'view' 'sensor')
  BinaryChoiceController ()
  FormMenuController ()
  MouseMenuController ('redButtonMenu' 'redButtonMessages'
  'yellowButtonMenu' 'yellowButtonMessages' 'blueButtonMenu'
  'blueButtonMessages')
    BitEditor ('scale' 'squareForm' 'color')
    FormEditor ('form' 'tool' 'grid' 'togglegrid' 'mode'
    'previousTool' 'color' 'unNormalizedColor' 'xgridOn'
    'ygridOn' 'toolMenu' 'underToolMenu')
    ScreenController ()
    ScrollController ('scrollBar' 'marker' 'savedArea')
      ListController ()
        LockedListController ()
          ChangeListController ()
        SelectionInListController ()
      ParagraphEditor ('paragraph' 'startBlock' 'stopBlock'
      'beginTypeInBlock' 'emphasisHere' 'initialText'
      'selectionShowing' 'currentFont' 'echoLocation' 'echoForm')
        StringHolderController ('isLockingOn')
          ChangeController ()
          FillInTheBlankController ()
            CRFillInTheBlankController ()
          ProjectController ()
          TextCollectorController ()

```
            TextController ()
              CodeController ()
                AlwaysAcceptCodeController ()
                OnlyWhenSelectedCodeController ()
          StandardSystemController ('status' )
            NotifierController ()
        NoController ()
        SwitchController ('selector' 'arguments' 'cursor' )
          IndicatorOnSwitchController ()
          LockedSwitchController ()
    ControlManager ('scheduledControllers' 'activeController'
    'activeControllerProcess' 'screenController' )
    Delay ('delayDuration' 'resumptionTime' 'delaySemaphore'
    'delayInProgress' )
    DisplayObject ()
      DisplayMedium ()
        Form ('bits' 'width' 'height' 'offset' )
          Cursor ()
          DisplayScreen ()
      DisplayText ('text' 'textStyle' 'offset' 'form' )
        Paragraph ('clippingRectangle' 'compositionRectangle'
        'destinationForm' 'rule' 'mask' 'marginTabsLevel'
        'firstIndent' 'restIndent' 'rightIndent' 'lines'
        'lastLine' 'outputMedium' )
          TextList ('list' )
      InfiniteForm ('patternForm' )
      OpaqueForm ('figure' 'shape' )
      Path ('form' 'collectionOfPoints' )
        Arc ('quadrant' 'radius' 'center' )
          Circle ()
          Curve ()
          Line ()
          LinearFit ()
          Spline ('derivatives' )
    Explainer ('class' 'selector' 'instance' 'context' 'methodText' )
    FileModel ('fileName' )
      FileList ('list' 'myPattern' 'isReading' 'fileMenu' )
    FormButtonCache ('offset' 'form' 'value' 'initialState' )
    InputSensor ('keyboardMap' )
    InputState ('x' 'y' 'bitState' 'lshiftState' 'rshiftState'
    'ctrlState' 'lockState' 'metaState' 'keyboardQueue' 'deltaTime'
    'baseTime' 'timeProtect' )
    Inspector ('object' 'field' )
      ContextInspector ('tempNames' )
      DictionaryInspector ('ok' )
    InstructionStream ('sender' 'pc' )
      ContextPart ('stackp' )
        BlockContext ('nargs' 'startpc' 'home' )
        MethodContext ('method' 'receiverMap' 'receiver' )
      Decompiler ('constructor' 'method' 'instVars' 'tempVars'
      'constTable' 'stack' 'statements' 'lastPc' 'exit' 'lastJumpPc'
```

'lastReturnPc' 'limit' 'hasValue')
  InstructionPrinter ('stream' 'oldPC')
KeyboardEvent ('keyCharacter' 'metaState')
Link ('nextLink')
  Process ('suspendedContext' 'priority' 'myList')
Magnitude ()
  Character ('value')
  Date ('day' 'year')
  LookupKey ('key')
    Association ('value')
  MessageTally ('class' 'method' 'tally' 'receivers')
  Number ()
    Float ()
    Fraction ('numerator' 'denominator')
    Integer ()
      LargeNegativeInteger ()
      LargePositiveInteger ()
      SmallInteger ()
  Time ('hours' 'minutes' 'seconds')
Message ('selector' 'args')
MethodDescription ('status' 'whichClass' 'selector')
ParseNode ('comment')
  AssignmentNode ('variable' 'value')
  BlockNode ('arguments' 'statements' 'returns' 'nArgsNode'
  'size' 'remoteCopyNode' 'sourceRange' 'endPC')
  CascadeNode ('receiver' 'messages')
  DecompilerConstructor ('method' 'instVars' 'nArgs'
  'literalValues' 'tempVars')
  Encoder ('scopeTable' 'nTemps' 'supered' 'requestor'
  'class' 'literalStream' 'selectorSet' 'litIndSet' 'litSet'
  'sourceRanges' 'lastTempPos')
  LeafNode ('key' 'code')
    LiteralNode ()
    SelectorNode ()
    VariableNode ('name' 'isArg')
  MessageNode ('receiver' 'selector' 'precedence' 'special'
  'arguments' 'sizes' 'pc')
  MethodNode ('selectorOrFalse' 'precedence' 'arguments'
  'block' 'literals' 'primitive' 'encoder' 'temporaries')
  ReturnNode ('expr' 'pc')
ParseStack ('position' 'length')
Point ('x' 'y')
PopUpMenu ('labelString' 'font' 'lineArray' 'frame' 'form'
'marker' 'selection')
  ActionMenu ('selectors')
ProcessHandle ('process' 'controller' 'interrupted'
'resumeContext' 'proceedValue')
ProcessorScheduler ('quiescentProcessLists' 'activeProcess')
Rectangle ('origin' 'corner')
  CharacterBlock ('stringIndex' 'character')
  Quadrangle ('borderWidth' 'borderColor' 'insideColor')

RemoteString ('sourceFileNumber' 'filePositionHi'
'filePositionLo')
Scanner ('source' 'mark' 'hereChar' 'aheadChar' 'token'
'tokenType' 'currentComment' 'buffer' 'typeTable')
  ChangeScanner ('file' 'chunkString')
  Parser ('here' 'hereType' 'hereMark' 'prevToken'
  'prevMark' 'encoder' 'requestor' 'parseNode' 'failBlock'
  'lastTempMark' 'correctionDelta')
SharedQueue ('contentsArray' 'readPosition' 'writePosition'
'accessProtect' 'readSynch')
Stream ()
  PositionableStream ('collection' 'position' 'readLimit')
    ReadStream ()
    WriteStream ('writeLimit')
      ReadWriteStream ()
        ExternalStream ()
          FileStream ('name' 'directory' 'mode' 'fileDescriptor'
          'filePosition' 'fileMode')
            FileDirectory ()
  Random ('seed')
StrikeFont ('xTable' 'glyphs' 'name' 'stopConditions'
'type' 'minAscii' 'maxAscii' 'maxWidth' 'strikeLength'
'ascent' 'descent' 'xOffset' 'raster' 'subscript'
'superscript' 'emphasis')
StringHolder ('contents' 'isLocked')
  ChangeList ('listName' 'changes' 'selectionIndex' 'list'
  'filter' 'removed' 'filterList' 'filterKey' 'changeDict'
  'doItDict' 'checkSystem' 'fieldList')
  FillInTheBlank ('actionBlock' 'actionTaken')
  Project ('projectWindows' 'projectChangeSet'
  'projectTranscript' 'projectHolder')
  TextCollector ('entryStream')
  Workspace ('localVariableDictionary')
Switch ('on' 'onAction' 'offAction')
  Button ()
  OneOnSwitch ('connection')
SyntaxError ('class' 'badText' 'processHandle')
SystemTracer ('map' 'refcts' 'file' 'holder' 'writeDict'
'maxOop' 'specialObjects' 'initialProcess' 'ot' 'bank' 'addr')
  TekSystemTracer ()
TekFileStatus ()
TekSystemCall ('operationType' 'operation' 'D0In' 'D0Out' 'D1In'
'D1Out' 'D2In' 'A0In' 'A0Out' 'errno')
TextStyle ('fontArray' 'lineGrid' 'baseline' 'alignment'
'firstIndent' 'restIndent' 'rightIndent' 'tabsArray'
'marginTabsArray' 'outputMedium')
UndefinedObject ()
View ('model' 'controller' 'superView' 'subViews'
'transformation' 'viewport' 'window' 'displayTransformation'
'insetDisplayBox' 'borderWidth' 'borderColor' 'insideColor'
'boundingBox' 'selectionSelected')

BinaryChoiceView ()
DisplayTextView ('rule' 'mask' 'editParagraph' 'centered')
FormMenuView ()
FormView ('rule' 'mask')
  FormHolderView ('displayedForm')
ListView ('list' 'selection' 'topDelimiter' 'bottomDelimiter'
'lineSpacing' 'isEmpty')
  ChangeListView ()
  SelectionInListView ('itemList' 'printItems' 'oneItem'
  'partMsg' 'initialSelectionMsg' 'changeMsg' 'listMsg'
  'menuMsg')
StandardSystemView ('labelFrame' 'labelText'
'isLabelComplemented' 'savedSubViews' 'minimumSize'
'maximumSize' 'windowForm' 'windowFormFlag')
  BrowserView ()
  InspectorView ()
  NotifierView ('contents')
StringHolderView ('displayContents')
  FillInTheBlankView ()
  ProjectView ()
  TextCollectorView ()
  WorkspaceView ()
SwitchView ('complemented' 'label' 'selector'
'keyCharacter' 'highlightForm' 'arguments')
  BooleanView ()
TextView ('partMsg' 'acceptMsg' 'menuMsg')
  CodeView ('initialSelection')
    OnlyWhenSelectedCodeView ('selectionMsg')
WindowingTransformation ('scale' 'translation')

APPENDIX B

SMALLTALK INTERNAL CHARACTER CODES

Section 8 of the 4404 AIS Reference Manual contains a table of Event Manager Key Codes. These raw key codes are mapped by the InpuiSensor class to an internal representation. This means that "shifted s" has an internal value that is different from an "unshifted s", which is also different from a "control s" and a "control shifted s". The internal values are the ones used by the ParagraphEditor.

Alphanumeric Keys

Table B-1 shows the Smalltalk Internal Character Code meanings for the main part of the keyboard — the "alphanumeric keys."

In this table, control characters are represented by the standard two- or three-letter abbreviations, given in ANSI X3.4 and ISO 646. Special symbols are represented by the four-character codes assigned to those symbols in ISO 6937. These meanings of these four-character codes are given in nearby notes.

Table B-1

Smalltalk Internal Character Codes Standard North American Keyboard															
Row 1 Keys (Mode)	{ [	! 1	@ 2	# 3	$ 4	% 5	^ 6	& 7	* 8	(9	) 0	SP09 SP10	+ =	}]	RUB
Unshifted	91	49	50	51	52	53	54	55	56	57	48	45	61	93	127
Shifted	123	33	64	35	36	37	94	38	42	40	41	95	43	125	127
Ctrl	132	136	144	143	128	130	129	131	180	149	135	137	6	29	127
Ctrl-Shifted	249	223	208	207	192	191	30	195	244	213	0	31	14	29	127
Row 2 Keys (Mode)	ESC	¯ I	Q	W	E	R	T	Y	U	I	O	P	` \	BSP	LF
Unshifted	27	124	113	119	101	114	116	121	117	105	111	112	92	8	10
Shifted	27	126	81	87	69	82	84	89	85	73	79	80	96	8	10
Ctrl	27	133	17	23	5	18	20	25	21	150	15	16	28	8	10
Ctrl-Shifted	27	134	203	209	194	239	240	242	197	214	216	202	28	8	10
Row 3 Keys (Mode)	TAB	A	S	D	F	G	H	J	K	L	: ;	* ´	RTN		
Unshifted	9	97	115	100	102	103	104	106	107	108	59	39	13		
Shifted	9	65	83	68	70	71	72	74	75	76	58	34	13		
Ctrl	9	1	19	4	6	7	8	10	11	12	3	138	13		
Ctrl-Shifted	9	212	211	196	226	241	243	229	200	217	3	219	13		
Row 4 Keys (Mode)	Z	X	C	V	B	N	M	< ,	> .	? /					
Unshifted	122	120	99	118	98	110	109	44	46	47					
Shifted	90	88	67	86	66	78	77	60	62	63					
Ctrl	26	24	3	22	2	14	13	1	18	27					
Ctrl-Shifted	231	215	228	198	230	245	246	218	233	203					
Row 5 Keys (Mode)	SPC														
Unshifted	32														
Shifted	32														
Ctrl	32														
Ctrl-Shifted	32														
Notes:	SP09: "low line" SP10: hyphen or minus sign														

B.1 Numeric Pad Keys

The numeric pad is located to the right of the main set of alphanumeric keys.

Table B-2

Smalltalk Internal Character Codes Standard North American Keyboard														
Key Pad Name Mode	0	1	2	3	4	5	6	7	8	9	.	,	-	ENT
Unshifted	48	49	50	51	52	53	54	55	56	57	46	44	45	27
Shifted	181	182	183	184	185	186	187	188	189	190	175	177	176	178
Ctrl	255	255	255	255	255	255	255	255	255	255	255	255	255	255
Ctrl-Shifted	255	255	255	255	255	255	255	255	255	255	255	255	255	255

B.2 Joydisk Keys

The joydisk is located to the upper left of the main set of alphanumeric keys.

Table B-3

Smalltalk Internal Character Codes Standard North American Keyboard				
Joydisk Key Name Mode	Up	Down	Right	Left
Unshifted	255	255	255	255
Shifted	255	255	255	255
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255

B.3 Function Keys

The function keys F1-F12 are grouped in three groups of four keys and are located in a row above both the alphanumeric keys and the numeric key pad.

Table B-4

Smalltalk Internal Character Codes Standard North American Keyboard				
Function Key Name Mode	F1	F2	F3	F4
Unshifted	151	152	153	154
Shifted	163	164	165	166
Ctrl	145	146	147	255
Ctrl-Shifted	255	255	255	255
Function Key Name Mode	F5	F6	F7	F8
Unshifted	155	156	157	158
Shifted	167	168	169	170
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255
Function Key Name Mode	F9	F10	F11	F12
Unshifted	159	160	161	162
Shifted	171	172	173	174
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255

B.4 Special Function Keys

There are only two special function keys on the Pegasus keyboard. One is the "up-arrow/left-arrow" key in the upper left corner of the main key area, while the other is the BREAK key in the lower right corner of the main key area.

Table B-5

Default ANSI Meanings of Special Function Keys Standard North American Keyboard		
Function Key Name	↑	BREAK
Mode	←	
Unshifted	94	179
Shifted	95	179
Ctrl	30	179
Ctrl-Shifted	148	179

INDEX

Index

Application Note

SUBTASK SUPPORT
AND MANAGEMENT

Support for subtasks created by the operating system is available in the Smalltalk-80 system. Although Smalltalk has its own processes, Smalltalk can also create and communicate with subtasks created by the operating system. Smalltalk processes provide independent paths of control within Smalltalk. All Smalltalk processes share access to the same set of objects. Operating system subtasks provide access to other executable programs. These subtasks are useful for running OS utilities and commands, as well as communicating with applications and programs written in other languages. Subtasks can be executed without leaving the Smalltalk environment. Protocol supporting subtasks is found in the classes **Subtask, Pipe, PipeStream** and its subclasses. The objective of this support is to make the job of creating, running and communicating with the subtasks straightforward. Interfaces to OS signals, program parameters, environment variables, and subtask priorities are also supported.

OVERVIEW OF SUBTASKS AND PIPES

The operating system on the 4404 supports multi-tasking. With Smalltalk's interface to the operating system via system calls, it is possible for Smalltalk to utilize this facility. Multi-tasking is achieved by spawning new tasks. A newly spawned task is called a child task or a subtask. The original task is referred to as the parent task. The child task is a "copy" of the parent task -- it shares memory and other resources with the parent task. Since only one task can execute at a time, cpu time is also shared, initially in a predetermined fashion. Common practice is for the spawned child task to perform some chore, and then report back to the parent task. After reporting, the child task disappears. This disappearance is known as the termination of the child task. A parent may chose to relinquish use of the CPU until a subtask terminates. It does this by an operation called waiting. While waiting, the parent task is blocked and cannot do anything else until the child task terminates.

The child task's chore is often accomplished by finding some other program to do the work. The use of this other program is known as an exec operation (for execute). In an exec operation, the original spawned task "turns itself into" the other program, so the other program becomes the child task.

Often times, a parent task may want to communicate with a child task. Information can be sent to and from the child task by using pipes, however, each pipe can send information in only one direction. If communication in two directions is desired, two pipes must be used. Pipes are similar to files with two critical differences.

- Files can be reopened many times. Pipes can only be opened once. Once a pipe is closed it is gone.

- Files can be reset and repositioned. It is not possible to reposition a pipe.

Usually the parent task creates a pipe. Each end of the pipe is assigned a file descriptor, one for reading and one for writing. When a subtask is created it inherits these open file descriptors. The parent task remembers one file descriptor, the one which is appropriate for its direction of communication. For example, if the parent task wants to send information to the child, the parent remembers the file descriptor for writing. Since the parent will not be using the reading end of the

pipe, it should close this unused end. The child task must also remember the appropriate file descriptor and close the file descriptor corresponding to the unused end of the pipe. Neglecting to close these unused pipe file descriptors might mean the task could run out of file descriptors, since there is a limit of 32 open file descriptors per task.

Sometimes it is not possible for the child task to know that it should use the pipe's file descriptors for reading and writing. For instance, the child task might want to exec a program that writes on standard output. Even if the program were aware of the use of pipes, it may not be possible for the program to modify itself to use the pipe's file descriptor for writing. In this case, it is possible for the child task to redirect its I/O by mapping its pipe descriptors to known file descriptors. (This mapping is accomplished through the use of the **dups** system call. See the *4404 Reference Manual* for more details.) Once a pipe's file descriptor is mapped, it becomes obsolete and should be closed. For example, the child task may want to write to the pipe, but the program is designed so write operations go to standard output. The write file descriptor of the pipe must be mapped to standard output's file descriptor (1), and the write file descriptor should be closed. The effect of the mapping in this example is for write operations on standard output from the child task to be performed on the write end of the pipe instead.

CLASS DESCRIPTIONS

Instances of the class **Subtask** represent operating system child tasks. The process of creating an instance of this class includes specifying an executable program and, optionally, arguments to the program and environmental variables. A block representing code to be executed by the child task can also be specified. **Subtask** contains protocol for invoking, terminating, and waiting for a child task. The invocation of a child task may include modification of the child's priorities and environment variables. The metaclass contains protocol for managing these child tasks.

The class **Pipe** represents an operating system pipe. Instance creation causes a pipe to be created with two open file descriptors, one for each end of the pipe. Protocol exists for mapping ends of the pipe to arbitrary file descriptors.

The class **PipeStream**, a subclass of **ExternalStream**, is an abstract class. Since instances of **ExternalStream** are positionable and pipes are not positionable, one of **PipeStream's** purposes is to provide protocol for filtering out inappropriate inherited methods. It also contains a method for closing a **PipeStream** or one of its subclasses, which also closes its associated pipe file descriptor. Protocol also exists for mode changes to binary or text. **PipeStream** has two subclasses, **PipeReadStream** and **PipeWriteStream**. Each of these subclasses is created by opening on an instance of **Pipe**. Each has protocol appropriate for its function: **PipeReadStream** has protocol for streaming over data read from a pipe. **PipeReadStream** buffers its data from the pipe. However, **PipeWriteStream** does not buffer its data. Both of these classes support non-homogeneous accesses, that is, reading or writing different sized pieces of data. These classes inherit higher

level protocol involving:

- Padding
- Accessing strings, numbers and words
- **fileIn** and **fileOut**

SUBTASK EXAMPLES

Here is an example which uses **Subtask**. Assume the existence of an executable file **/bin/simpleUtility**, a program with no input, output, or arguments.

```
executeSimple
      "Execute a pretend program."
      | task |
      task ← Subtask fork: '/bin/simpleUtility' then: [].
      task start.
      task waitOn.
      task release.
```

The method **fork:then:** creates an instance of **Subtask**. This object, assigned to the variable **task**, contains all the information needed to create an OS subtask to execute the program **/bin/simpleUtility**. However, an actual subtask is not created by this method. The method **start** issues the system calls **vfork** and **exec** to create and run the subtask. The method **waitOn** instructs the currently executing Smalltalk process to wait for the subtask to terminate. **Release** discards the **Subtask** object.

Here is a somewhat more complicated example:

```
execSystemUtility: aCommand
      | pipe task inputSide resultOfProgram |
      pipe← Pipe new.
      task ← Subtask fork: aCommand then: [
                  pipe mapWriteTo: 1.
                  pipe mapWriteTo: 2.
                  pipe closeWrite; closeRead].
      task start.
      pipe closeWrite.
      inputSide ← PipeReadStream openOn: pipe.
      resultOfProgram ← inputSide contentsOfEntireFile.

      task waitOn.
      inputSide close.
      task release.
      ↑resultOfProgram
```

In this method, **execSystemUtility:**, a pipe is created to establish one-way communication with the subtask. (Two pipes are required for two-way communication.) The code in the block is executed by the subtask after the vfork

system call and before the exec system call. All the rest of the code in
execSystemUtility: is executed by the parent task.

Pipe connections in the child task are established in the block. In this case, the
child task's standard output (file descriptor 1) and standard error (file descriptor 2)
are redirected to the pipe through the use of the **mapWriteTo:** method. When the
child task writes to standard output or standard error, this mapping causes the
write operations to be directed to the write side of the pipe. Since the write end of
the pipe has been redirected, it is a good idea to close the write end with the
message **closeWrite**. In addition to closing redirected ends of the pipe in the child
task, unused ends of the pipe should be closed in both the parent and child tasks. In
this case, the pipe read end is unused in the subtask, and the pipe write end is
unused in the parent task.

The net affect of all this closing and mapping is that the child task (whose code is
executed in the block) closes the read side of the pipe because it is unused and
closes the write side of the pipe because it has mapped the write side to standard
output and standard error. The parent task closes its unused end of the pipe, which
is the write side. The parent task also creates a Smalltalk object for reading from
the pipe, an instance of **PipeReadStream** called **inputSide**. **InputSide** inherits
protocol from **PipeStream** and consequently **ExternalStream**. Although other
methods may be used to read from the pipe, here, the method
contentsOfEntireFile is used to read all the data from the pipe, and the pipe is
closed after use.

The following method, found in **execSystemUtility:withArgs: TekSystemCall
class**, in addition to havinf the same functionality as the method immediately above,
also has error checking and passes arguments to the executable program,
aCommand.

```
execSystemUtility: aCommand withArgs: anOrderedCollection
      | pipe task inputSide resultOfProgram |
      pipe ← Pipe new.
      task ← Subtask
                        fork: aCommand
                        withArgs: anOrderedCollection
                        then:
                                [pipe mapWriteTo: 1.
                                pipe mapWriteTo: 2.
                                pipe closeWrite; closeRead].
      task start isNil
            ifTrue:
                  [pipe closeWrite; closeRead.
                  self error: 'Cannot execute ' , aCommand].
      pipe closeWrite.
      Cursor execute
            showWhile:
                  [inputSide ← PipeReadStream openOn: pipe.
                  resultOfProgram ← inputSide
                  contentsOfEntireFile].
      task waitOn.
      inputSide close.
      task abnormalTermination ifTrue:
            [self error: 'Error from system utility: ' ,
                  (resultOfProgram copyUpTo: Character cr)].
      task release.
      ↑resultOfProgram
```

This method contains code to

- Pass arguments to the program in the form of an **OrderedCollection**.

- Check for failure of the child task (**task start isNil**).

- Test for abnormal termination of the child task.

Failure of the child task necessitates the closing of any pipes created for use in the subtask. The parent task which creates these **Pipes** is responsible for closing them. Neglecting to close these pipes might mean the Smalltalk parent task could run out of file descriptors.

Examples Using `execSystemUtility:withArgs:`

Here are some examples that demonstrate how to use
`execSystemUtility:withArgs:`.

To execute a program with arguments:

```
fileName ← 'timingData'.
flags ← '+sa'.
TekSystemCall
        execSystemUtility: '/bin/dir'
        withArgs: (OrderedCollection with: fileName with: flags).
```

The next example executes a shell with a **+c** option. The **+c** option tells the shell to read the rest of the arguments as a command to itself. The effect is a directory listing with the shell providing wildcard expansion.

```
pattern ← '/smalltalk/de*'.
nameList ← TekSystemCall
                execSystemUtility: '/bin/shell'
                withArgs: (OrderedCollection
                                with: '+c'
                                with: '/bin/dir +s ' , pattern)
```

Besides taking advantage of the shell's wildcard expansions, you can also use aliases which are stored in the **.shellhistory** file. (See the *4404 Reference Manual* — the **shell** command for more details.)

```
TekSystemCall
        execSystemUtility: '/bin/shell'
        withArgs: (OrderedCollection with: '+c' with: 'df')
```

where **df** is an alias for **free /dev/disk**.

To execute a program with no arguments substitute an empty **OrderedCollection** for the second argument.

```
TekSystemCall
        execSystemUtility: '/bin/date'
        withArgs: OrderedCollection new.
```

Environment Variables

The Smalltalk-80 system's interface to subtasks also supports environment variables. (See the *4404 Reference Manual* for more details.) In general, when a program is invoked, the operating system passes arguments and environment variables to the program. Standard environment variables include **HOME** — a home directory specification and **PATH** — a search path specification. Environments are a way to pass information by name. This can be viewed as setting a context for execution. Instances of subtask are created with a default environment, the environment with which Smalltalk was invoked. The method **Subtask class copyEnvironment** answers a copy of the default environment. This copy is in dictionary format for easy modification. The method **Subtask environment:** assigns an environment to the **Subtask** instance which passes it to the executed program. Here is an example of use of a modified environment which specifies that **/smalltalk** is the current **HOME** directory.

```
execProgram: aCommand
      | task env |
      task ← Subtask
                      fork: aCommand
                      then: [].
      env ← Subtask copyEnvironment.
      env at: #HOME put: '/smalltalk'.
      task environment: env.
      task start isNil
            ifTrue:
                  [self error: 'Cannot execute ' , aCommand].
      task waitOn.
      task abnormalTermination ifTrue: [self error: 'Error from ' , aCommand].
      task release.
```

Signals

Operating system signals (colloquially referred to as *interrupts*) can be intercepted, ignored, or set to a default action by using protocol in **TekSystemCall** class. Usually, the default action upon receipt of an interrupt is task termination. (See the *4404 Refence Manual* — the **int** command and **cpint** system call for more details.) Sometimes it is desirable for the child task to intercept, or modify, its reaction to an interrupt. Protocol to modify these reactions can be added to the block which is an argument to the **Subtask** instance creation methods. In our next example, **ScreenController forkOSShell**, the method **fork:withArgs:then:** is passed a block which modifies some of these reactions. Code in this block is executed by the child task only. The reaction to several interrupts is modified with the method **setInterrupt:to:** in both the parent and child task. Here is a simplified and stripped down copy of the method **ScreenController forkOSShell**.

```
forkOSshell
      "Simplified for example."

      | location task oldSIGHUPValue oldSIGINTValue oldSIGQUITValue
            sysCall oldSIGTERMValue |
      oldSIGHUPValue ← TekSystemCall setInterrupt: 1 to: 1.
      oldSIGINTValue ← TekSystemCall setInterrupt: 2 to: 1.
      oldSIGQUITValue ← TekSystemCall setInterrupt: 3 to: 1.
      oldSIGTERMValue ← TekSystemCall setInterrupt: 11 to: 1.
      task ← Subtask fork: '/bin/shell'  withArgs:
            (OrderedCollection with: '+1')
            then: [
            TekSystemCall setInterrupt: 1 to: 0.
            TekSystemCall setInterrupt: 2 to: 0.
            TekSystemCall setInterrupt: 3 to: 0.
            TekSystemCall setInterrupt: 11 to: 0.
            sysCall ← TekSystemCall terminalOn.
            sysCall value].
      error ← task start.
      error isNil
            ifFalse: [task absoluteWait.
                  task release].

      TekSystemCall setInterrupt: 1 to: oldSIGHUPValue.
      TekSystemCall setInterrupt: 2 to: oldSIGINTValue.
      TekSystemCall setInterrupt: 3 to: oldSIGQUITValue.
      TekSystemCall setInterrupt: 11 to: oldSIGTERMValue.
      sysCall ← TekSystemCall terminalOff.
      sysCall value.
      ScheduledControllers restore.
      error isNil ifTrue: [↑self error: 'Cannot fork shell']
```

Interrupt action is modified in both the parent and child tasks by using the method
setInterrupt:to:, which returns the previous action for that interrupt. First, the
parent interrupt actions are saved in temporary variables while setting interrupt
action to 1, which means to ignore the interrupt. In the subtask, these same
interrupts are reset to the default action, by making the interrupt action 0. After the
subtask has completed, the interrupts in the parent task are set back to their
original values. This subtask runs by using the protocol previously described, but
the parent task waits for the child task to terminate by using the method
absoluteWait. This method actually shuts down the Smalltalk parent task so it
receives no time slice from the operating system scheduler. This strategy of waiting
makes the subtask more efficient because the parent task cannot steal any
processing power. However, Smalltalk cannot run until the child task has
terminated. **AbsoluteWait** is not appropriate for any subtask that depends on the
Smalltalk user interface.

Priorities and Two Way Communication

Subtasks can be made to run more efficiently by changing their priorities. Sending the message **priority:** to an instance of **Subtask** modifies the invocation of a subtask so that it runs at the designated priority. Here is the definition of the method **Subtask priority:**.

```
priority: aPriority
    "Set the priority of the subtask.  Acceptable values range from 0 to
    25, zero being the highest and 25 being the lowest."

    aPriority < 0 | (aPriority > 25)
        ifTrue: [self error: 'Unacceptable priority value'].
    priority ← aPriority
```

The Smalltalk parent task initially has a priority of 10. If the child task is created with a higher priority than the parent task, it has a potential of taking control of the CPU. A higher priority task will not relinquish the CPU to a lower priority task unless the higher one is blocked or terminates. The next example increases the child task's priority and uses two pipes for two way communication. It is known that the child task in this case will be blocked while waiting for input, so the parent task will have a chance to run. The class **ShellInterface** creates a view that communicates with a shell (**/bin/script**) interactively. (Note that this class is not contained in the **standardImage** but defined in the file **/smalltalk/fileIn/Examples-Subtasking** .) After an instance of this class is created, it is initialized with the following method.

```
initialize
      | pipeIn pipeOut sysCall |
      command ← 'command'.
      result ← '' asText.
      pipeIn ← Pipe new.
      pipeOut ← Pipe new.
      shellTask ← Subtask fork: '/bin/script'
                          then:
                                  [FileStream releaseStdRefs.
                                  pipeIn mapWriteTo: 1.
                                  pipeIn mapWriteTo: 2.
                                  pipeIn closeWrite.
                                  pipeIn closeRead.
                                  pipeOut mapReadTo: 0.
                                  pipeOut closeRead.
                                  pipeOut closeWrite].
      shellTask enhancedPriority.
      shellTask start isNil
            ifTrue:
                  [pipeIn closeWrite; closeRead.
                  pipeOut closeWrite; closeRead.
                  self error: 'Cannot execute a shell'].

      pipeOut closeRead.
      pipeIn closeWrite.
      shellIn ← PipeReadStream openOn: pipeIn.
      shellOut ← PipeWriteStream openOn: pipeOut
```

This method uses two pipes for two way communication. It also assigns the highest possible priority to the subtask with the message **enhancedPriority**. When the view is closed, each instance of **ShellInterface** cleans up with the release method.

```
release
      | sysCall |
      "Close pipes for interactive shell"
      shellOut isValid ifFalse: [↑self].
      shellIn close.
      shellOut close.
      shellTask kill.
      shellTask waitOn.
      shellTask release.

      TekSystemCall terminalOff value.
      TekSystemCall cursorOn value.
      Cursor cursorLink: true.
      Display enableCursorPanning.
      Display enableJoydiskPanning.         .
      Display setNormalVideo.
```

The shell subtask, **shellTask**, is terminated with the **kill** message. The parent
task waits for the shell subtask to terminate and then releases the instance of
Subtask. Other methods are invoked to reset states in case a subtask of the shell
has modified the display.

THE DETAILS OF STARTING A SUBTASK

Here is the **start** method taken from the class **Subtask**. This method contains low level details of how a subtask is actually exec'ed.

```
start
        "Start the receiver by executing a vfork, code to set up
         the child task (mainly communication and signal processing),
         and exec'ing the program.  If the exec fails terminate the
         child task.  The child task will inherit the priority of
         the Smalltalk task."

        | forker execer task |
        forker ← TekSystemCall vfork.
        environment isNil
                ifTrue: [execer ← TekSystemCall exec: program with: args]
                ifFalse: [execer ← TekSystemCall execve: program
                    withArgs: args withEnv: environment].

        forker value.
        initBlock value.
        self priority notNil ifTrue: [(TekSystemCall setpr: priority) value].
        FileStream closeExternalReferences.
        execer invoke
                ifFalse:
                        [(TekSystemCall term: 0) value.
                        self taskId: nil.
                        ↑nil].
        self taskId: execer DOOut.
        ScheduledSubtasks add: self.
        self criticalSection: [self status: #running].
```

Subtasks are run from Smalltalk by making two essential system calls. The **vfork** system call, **forker**, creates a child task which shares memory with the parent task. The **exec/execve** system call, **execer**, transforms the child task so it is no longer running Smalltalk, but is executing the specified binary file, **program**. **Execer** is created with either the **exec** system call (**exec:with:**) or with the **execve** system call (**execve:withArgs:withEnv:**), depending on whether an environment is passed to the binary program.

Before the **exec/execve** invocation (**execer value**), communications and signals must be set up. When an instance of **Subtask** is created, potentially, a block is passed as an argument containing code for setting up communications and signals in the child task. This block was stored in the instance variable, **initBlock**, which is evaluated at this point. Priorities are also assigned at this time, and files belonging to the parent task Smalltalk are closed with the expression **FileStream closeExternalReferences**. After the child task has been spawned and control returns to the parent, the task identification number is recorded by the parent task (**self taskId: execer DOOut**), the child task is added to the list of managed subtasks (**ScheduledSubtasks add: self**), and its status is recorded. If the **execer**

call fails, the child task terminates itself with the **term:** method and the parent task returns **nil**.

At the **vfork** invocation (**forker value**), control of the Smalltalk virtual machine transfers to the child task. The child task continues executing with no access to the keyboard or the mouse until the **execer** call is invoked. This means that the code from **forker value** to **execer invoke** is only executed by the child task. In addition,

```
(TekSystemCall term: 0) value.
```

is executed by the child task to terminate itself if the **execer** call fails. Then the parent task resumes control and executes the statements

```
self taskId: nil.
↑nil
```

to indicate failure of the **execer** call. The next statements, starting with **self taskId: execer DOOut**, are also executed by the parent task, but only if the **execer** call was successful. Whenever the child task is blocked or the child task terminates, control reverts back to the parent task. The parent task initially resumes control in the state left by the child task before the **forker** invocation.