GO Corporation

GO Technical Library

PenPoint™ Development Tools

PenPoint

PenPoint™

PenPoint™ Development Tools

GO CORPORATION

GO TECHNICAL LIBRARY

PenPoint Application Writing Guide provides a tutorial on writing PenPoint applications, including many coding samples. This is the first book you should read as a beginning PenPoint applications developer.

PenPoint Architectural Reference Volume I presents the concepts of the fundamental PenPoint classes. Read this book when you need to understand the fundamental PenPoint subsystems, such as the class manager, application framework, windows and graphics, and so on.

PenPoint Architectural Reference Volume II presents the concepts of the supplemental PenPoint classes. You should read this book when you need to understand the supplemental PenPoint subsystems, such as the text subsystem, the file system, connectivity, and so on.

PenPoint API Reference Volume I provides a complete reference to the fundamental PenPoint classes, messages, and data structures.

PenPoint API Reference Volume II provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

PenPoint User Interface Design Reference describes the elements of the PenPoint Notebook User Interface, sets standards for using those elements, and describes how PenPoint uses the elements. Read this book before designing your application's user interface.

PenPoint Development Tools describes the environment for developing, debugging, and testing PenPoint applications. You need this book when you start to implement and test your first PenPoint application.

PenPoint™

PenPoint™ Development Tools

GO CORPORATION

GO TECHNICAL LIBRARY

Addison-Wesley Publishing Company

Reading, Massachusetts ◆ Menlo Park, California ◆ New York
Don Mills, Ontario ◆ Wokingham, England ◆ Amsterdam
Bonn ◆ Sydney ◆ Singapore ◆ Tokyo ◆ Madrid ◆ San Juan
Paris ◆ Seoul ◆ Milan ◆ Mexico City ◆ Taipei

ISBN 0–201–60861–8

123456789–AL–9695949392

First printing, June 1992

Preface

PenPoint Development Tools provides detailed information on the tools and facilities available to developers who are writing programs for the PenPoint™ operating system. This volume includes descriptions of the other volumes in the PenPoint SDK (software development kit), how to run the PenPoint operating system on a desktop PC, how to use the PenPoint source code debugger, and how to use other miscellaneous tools.

This volume also contains a Master Index to the entire SDK documentation set.

Intended Audience i.1

PenPoint Development Tools is written for people who are designing and developing applications and other programs for the PenPoint operating system. This book addresses both those who are using high-level development tools (application generators and the like) and those who are writing programs that directly access the PenPoint APIs (application programmatic interfaces). However, parts 2 and 3 of this book will be more useful to those writing programs that access the PenPoint APIs.

What's Here i.2

PenPoint Development Tools is divided into several parts:

- *Part 1: Getting Started,* describes what is involved in developing PenPoint programs. It the describes the PenPoint documentation and how to run the PenPoint operating system on a desktop PC.
- *Part 2: Debugging PenPoint Programs,* describes the PenPoint source-level debugger, the PenPoint mini-debugger, and other debugging tools and techniques available to developers.
- *Part 3: Tools,* describes other tools that can assist you in developing or embellishing your application. These tools include DOS tools, a bit-map icon editor, a screen shot utility, and a font editor.

Other Sources of Information i.3

The *PenPoint Application Writing Guide* provides a tutorial on writing PenPoint applications. The tutorial is illustrated with several sample applications.

The *PenPoint Architectural Reference* groups the PenPoint classes into several functional areas and describes how to use these classes. The *PenPoint Architectural Reference* is divided into two volumes. The first volume describes the fundamental classes that all application developers will use; the second volume describes supplemental classes that application developers may, or may not, use.

The *PenPoint API Reference* is a set of "datasheets" that were generated from the PenPoint SDK header files. These datasheets contain information about all the messages defined by the public PenPoint classes. If you own the PenPoint SDK, you can also find the header files in the directory \PENPOINT\SDK\INC.

To learn how to use PenPoint, you should refer to the PenPoint user documentation. The user documentation is included with the PenPoint SDK, and is usually packaged with a PenPoint computer. The user documentation consists of these books:

- *Getting Started*, a primer on how to use PenPoint.
- *Using PenPoint*, a detailed book on how to use PenPoint to perform tasks and procedures.
- *Using GOWrite*, which helps users to develop more effective handwriting when using the GOWrite handwriting translation engine.

Type Styles in This Book i.4

To emphasize or distinguish particular words or text, we use different fonts.

Computerese i.4.1

We use fonts to distinguish two different forms of "computerese":

- C Language keywords and preprocessor directives, such as `switch`, `case`, `#define`, `#ifdef`, and so on.
- Functions, macros, class names, message names, constants, variables, and structures defined by PenPoint, such as **DPrintf()**, **msgListAddItem**, **clsList**, **stsBadParam**, P_LIST_NEW, and so on.

Although all these PenPoint terms use the same font, you should note that PenPoint has some fixed rules on the capitalization and spelling of messages, functions, constants, and types. By the spelling and capitalization, you can quickly identify the use of a PenPoint term.

- Classes begin with the letters "**cls**," for example **clsList**.
- Messages begin with the letters "**msg**," for example **msgNew**.
- Status values begin with the letters "**sts**," for example **stsOK**.
- Functions are mixed case with an initial upper case letter and are terminated with open and close parenthesis, for example **OSMemAvailable()**.
- Constants are mixed case with an initial lower case letter, for example **wsClipChildren**.
- Structures and types are all upper case (with underscores, when needed, to increase comprehension), for example, U32 or LIST_NEW_ONLY.

Code Listings i.4.2

Code listings and user-PC dialogs appear in a fixed-width font.

```
// Allocate, initialize, and record instance data.
//
StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
        s, Error);
pInst->>placeHolder = -1L;
ObjectWrite(self, ctx, &pInst);
```

Less significant parts of code listings are grayed out to de-emphasize them. You needn't pay so much attention to these lines, although they are part of the listing.

```
ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);
new.object.uid                  = clsTttApp;
new.object.key                  = 0;
new.cls.pMsg                    = clsTttAppTable;
new.cls.ancestor                = clsApp;
new.cls.size                    = SizeOf(P_TTT_APP_INST);
new.cls.newArgsSize             = SizeOf(APP_NEW);
new.appMgr.flags.stationery     = true;
new.appMgr.flags.accessory      = false;
strcpy(new.appMgr.company, "GO Corporation");
new.appMgr.copyright = "\213 1992 GO Corporation, All Rights Reserved.";
ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
```

Placeholders i.4.3

Anything you do *not* have to type in exactly as printed is generally formatted in italics. This includes C variables, suggested filenames in dialogs, and pseudocode in file listings.

Other Text i.4.4

The documentation uses *italics* for emphasis. When a part uses a significant term, it is usually emphasized the first time. If you aren't familiar with the term, you can look it up in the Glossary in the *PenPoint Application Writing Guide* or the index of the book.

DOS filenames such as \\BOOT\PENPOINT\APP are in small capitals. PenPoint file names, which can be upper and lower case, are shown in smaller type, such as \\My Disk\\Package Design Letter.

Book names such as *PenPoint Application Writing Guide* are in italics.

PENPOINT DEVELOPMENT TOOLS
CONTENTS

Part 1 / Getting Started

List of Figures

List of Tables

Chapter 1 / Welcome

Welcome to developing applications for the PenPoint™ operating system.

This manual has several aims:

- It tells you what you need to know before starting to develop applications for PenPoint.
- It tells you how the PenPoint SDK documentation is organized.
- It tells you the PC equipment you need to compile and run PenPoint programs.
- It tells you how to run PenPoint on a PC.

Development Options 1.1

There are two ways to develop applications on PenPoint:

- Using high-level development tools, such as form builders, 4GL languages, and scripting languages. These tools allow you to write applications without having to access the PenPoint APIs. Using high-level development tools, you can develop applications quickly without needing to learn many of the details of PenPoint. This is at the cost of some speed and flexibility.
- Using a compiled language (such as C) and the PenPoint APIs. This method allows you to write applications that access the PenPoint APIs directly. While the development time is longer, the resulting applications can be faster and more flexible.

Both options have their advantages and disadvantages; your choice of one over the other depends on several factors, including the type of problem you are solving, the existence of a development tool that can provide a solution to this problem, the time available to develop a solution, and your performance requirements.

Using High-Level Development Tools 1.1.1

Almost all high-level development tools run on PenPoint running on either tablet hardware or desktop PCs. If you use a desktop PC, you need:

- A desktop PC that runs PenPoint (see list of supported machines in Chapter 3).
- A digitizer tablet.
- A PenPoint SDK.
- The development tool you will be using to develop your applications.

If you use high-level development tools, you do not need all the extended SDK documentation. Most of the documentation is aimed at developers who are writing PenPoint applications using the PenPoint API.

Using the PenPoint APIs 1.1.2

So that you can start to get an idea of how you develop applications for PenPoint, here is what you do to create a PenPoint application:

1. Design Your Application. Consider the user interface that you need, how you intend to interact with other PenPoint applications, what devices you need to access, and how you intend to store your data.
2. Decide what existing classes you can use. PenPoint provides a rich set of classes that can do much of the work for your application. Your task is to decide which of these classes will serve you best. The task of the *PenPoint Architectural Reference* is to help you find the classes you need.
3. Decide what new classes you need to create (and their messages). Perhaps the PenPoint classes don't do exactly what you need. Look for the class that comes closest to your needs, then create your own class that inherits behavior from that class.
4. Write the application. PenPoint applications are written in the C language; the object oriented extensions are provided through function calls. Your application uses the PenPoint Application Framework, which performs much of the standard application tasks for you.
5. Compile, install, test, and debug. This is the classical cycle. A good strategy is to create an empty application that does no work, but still will appear as an empty page in the Notebook. Then you add the code that creates a new object or defines a new class, compile, install, test, and so on until you have a completed application.
6. Pretty up the application. When your application is nearly done, you should create icons, help notebook pages, quick help, stationery, and many other things that make your application "real."
7. Contact GO Developer Technical Support. Before you can release a PenPoint application, you need to register your classes with GO. This ensures that no other PenPoint developer will use your class's unique identifiers (UIDs).
8. Ship product, schedule celebration. Of course, don't forget the documentation.

This information is extracted from the *PenPoint Application Writing Guide*, which describes in detail how to create an application. However, before you get there, there are a number of concepts that we need to cover.

Clearly you need to have some familiarity with programming in the C language to write PenPoint applications. In addition to the C language proper, you should also be familiar with DOS C development environments and tools, such as MAKE.

Writing PenPoint application also requires some familiarity with object-oriented programming and techniques, although the *PenPoint Application Writing Guide* spends some time explaining concepts such as classes, objects, messages, inheritance, ancestors and so on.

On occasion, debugging PenPoint programs may require some familiarity with Intel assembly language.

Additional Reading 1.2

To develop applications that access the PenPoint APIs, you need to know how to program in the C language and you should be familiar with DOS C development environments. If you need further help, there are many good books available on these subjects.

For a description of the 80386 architecture and instruction set (which you need for debugging), see:

- *Intel 386DX Programmer's Reference Manual*, Intel.

Writing PenPoint applications definitely requires familiarity with object-oriented programming. Although the SDK documentation covers this topic to some extent, these books provide much more information on object-oriented programming:

- *Principles of Object Oriented Design*, Grady Booch, Redwood City, CA, The Benjamin/Cummins Publishing Co., 1991.
- *Object-Oriented Programming: An Evolutionary Approach*, 2nd edition, Brad J. Cox, Reading, MA, Addison-Wesley Publishing Company, 1991.
- *Object-Oriented Modeling and Design*, Rumbaugh, James, Michael Blaha, William Premerlani, Fredrick Eddy, and William Lorensen, Englewood Cliffs, NJ, Prentice-Hall, 1991.
- *Designing Object-Oriented Software*, Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener, Englewood Cliffs, NJ, Prentice-Hall, 1990.

Chapter 2 / The PenPoint Documentation

The PenPoint™ operating system is a rich operating system that makes most of its system-defined classes available to application developers. In fact, you *must* use some of the classes to create a PenPoint application.

However, with this embarassment of riches comes some difficulty in knowing exactly what you want or need from the operating system.

This chapter has two purposes. The first is to give you a guide to the PenPoint SDK documentation, its organization, and how we intend you to read it. In describing the documentation, we accomplish the second purpose, which is to give you an overview of the components and features of PenPoint.

If you plan to develop PenPoint applications using a high-level development tool, you may find this tour of the documentation useful. It can help you to decide whether you want to develop using a high-level development tool or the PenPoint APIs.

A Suggested Approach to Documentation 2.1

The point of the PenPoint developer documentation is to teach you:

- How to write applications.
- How to find your way around these many classes.

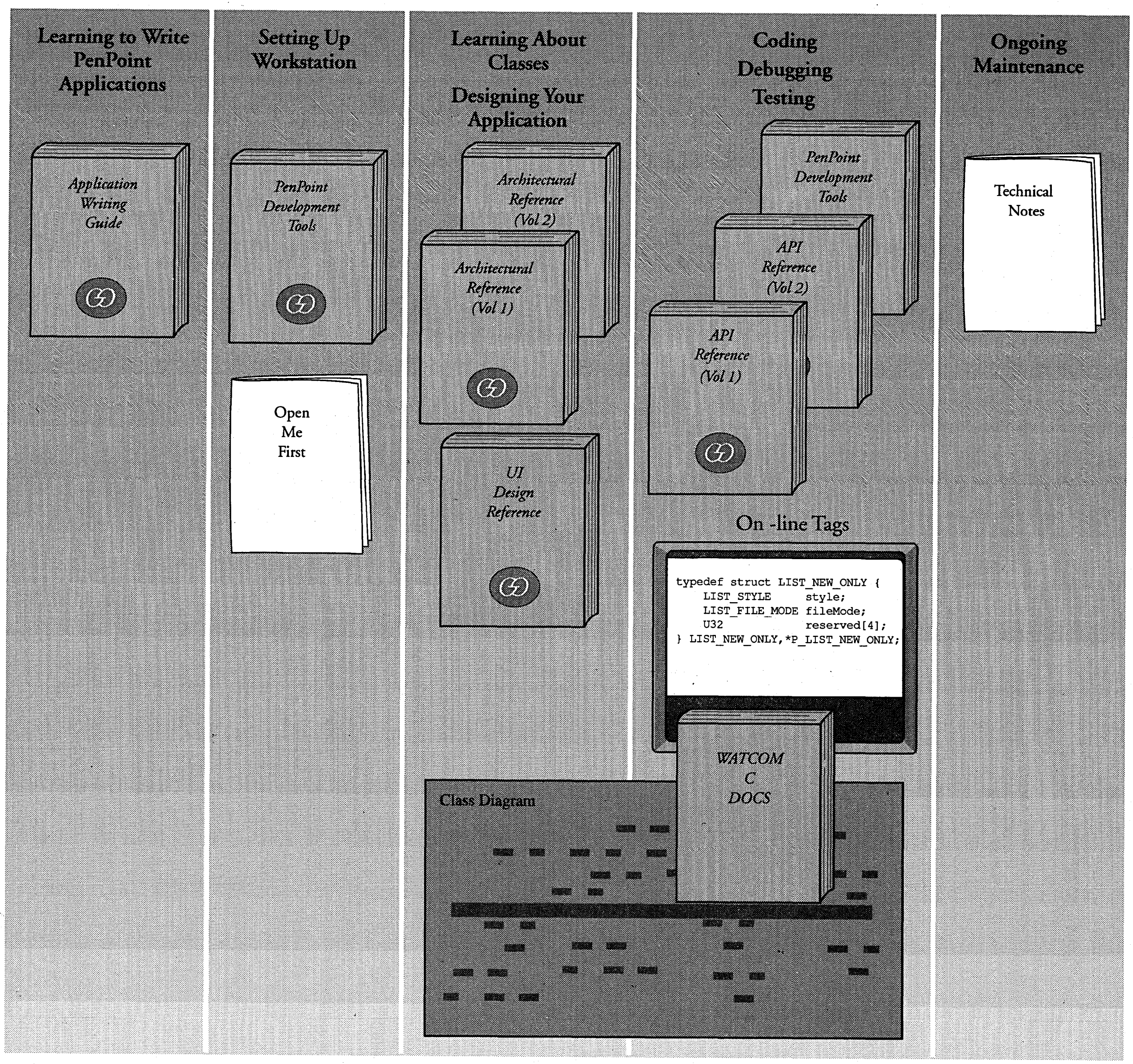
The manuals that make up the SDK documentation library are:

- *PenPoint Development Tools* (this manual)
- *PenPoint Application Writing Guide*
- *PenPoint UI Design Reference*
- *PenPoint Architectural Reference* (two volumes)
- *PenPoint API Reference* (two volumes)

The books *Getting Started with PenPoint*, *Using PenPoint*, and *Using GOWrite* teach you how to use the PenPoint operating system. These books are part of the user documentation set that accompanies a pen computer with the PenPoint operating system. They are not considered part of the developer documentation set.

This figure shows the PenPoint SDK documentation library and how we suggest you should use it.

Figure 2-1
Documents in the Software Development Kit



Feedback on Documentation 2.1.1

The technical documentation team is eager to get your comments and feedback on the documentation. There is a Reader's Comments Form in the back of each volume of the PenPoint SDK documentation. If you have suggestions, criticisms, or even compliments on the documentation, please let us know. It certainly helps us to know what we're doing right, in addition to what we're doing wrong.

You can fax us your Reader's Comment Form, and any marked-up pages, at (415) 345-9833.

Application Writing Guide

2.2

The *Application Writing Guide* provides a tutorial on how to write PenPoint applications.

System Overview provides an architectural view of the PenPoint operating system. The part describes the kernel, system, component, and application layers of PenPoint, and also touches on the Application Framework and the development environment.

Application Concepts presents applications from a conceptual point of view. This chapter describes most of the parts of an application that you must write, and also describes parts of applications that are provided for you by the PenPoint Application Framework.

A Developer's Checklist presents a checklist that you can use to ensure that your application is complete. The subsequent sections of the PenPoint Application Writing Guide describe in detail how to implement the parts of the checklist.

Designing Your Application discusses the points that you must consider when creating an application.

Compiling PenPoint Programs describes how to compile and link a PenPoint application.

Running PenPoint describes how to run PenPoint on a PC, and how to install programs in PenPoint.

A Simple Application introduces a minimal application. Through this application, you can see just how much the PenPoint Application Framework does for you.

Debugging describes the tools available to you to test and debug PenPoint Programs.

Creating Objects describes how you create instances of predefined PenPoint classes and use these objects in your application.

Using Windows describes how to use some of the most useful classes in PenPoint, the windows classes. The chapter describes how to create custom windows and presents additional information about using instance data.

Saving and Restoring Data describes how to save and restore data from your application.

Adding Classes describes how you create new classes and add them to PenPoint.

Refining the Application describes how to add polish to your application.

Releasing the Application describes the steps necessary to make your application available to other PenPoint users.

The *Appendix* describes the programs referred to in the previous chapters and other sample applications in \PENPOINT\SDK\SAMPLE and provides source code listings for some of these applications.

Glossary provides a list of terms used in the PenPoint SDK documentation.

There are more sample applications in the SDK than are described in *Application Writing Guide: Sample Code.* These sample applications are listed in Table 2-1.

Table 2-1

Sample Code in \PENPOINT\SDK\SAMPLE

Directory	What it Is	What it Demonstrates
ADDER	handwriting calculator	How to write insertion pads and translators for handwritten input.
BASICSVC	Basic service	The fundamental parts to a PenPoint service.
CALC	Calculator accessory	Separate "engine" and application objects, table layout, custom layout, labels, filing.
CLOCK	Clock accessory	Embedding, option sheets, modified application defaults, system timers.
CNTRAPP	App developed in ADC labs	Saving and restoring application state, memory-mapping state data.
EMPTYAPP	Minimal application	How the PenPoint application framework performs application boilerplate work.
HELLO	Graphical "Hello World"	Graphics, text, font scaling, subclassing clsWin.
HELLOTK	UI Toolkit "Hello World"	Custom layout, labels.
INPUTAPP	Pen input demo	Pen input, responding to input events.
MILSVC	PenPoint device driver	How to write a PenPoint MIL service (or device driver).
NPAPP	Note paper application	Using the NotePaper class that supports ink as a data type.
PAINT	Painting application	Using pixel maps, drawing contexts, pen input, and non-standard menu bars.
TEMPLTAP	Template application	A template for starting new applications.
TESTSVC	A test service	Compiling, installing, and running services.
TKDEMO	UI Toolkit demo	Using most of the classes in the UI toolkit (borders, labels buttons, fields, menus, custom layouts, table layouts, toolkit tables, notes, option sheets, etc.).
TTT	Tic-tac-toe application	Gestures, keyboard input, simple graphics, import/export, move/copy, undo, application menus, view-data model.
WRITERAP	Handwriting recognizer	menus, subclassing clsSPaper, using translator objects, constraining recognition.

PenPoint Development Tools 2.3

This volume, *PenPoint Development Tools*, provides a description of the PenPoint development environment and documents the tools you use when developing applications.

Getting Started describes the overall organization of the SDK documentation and how to run PenPoint on a PC. If you are developing an application using high-level development tools, this is probably all that you need to read.

Debugging PenPoint Programs describes the PenPoint debugging tools, including the source-level debugger (DB), the mini-debugger, and the System Log application.

Development Tools, describes DOS and PenPoint utilities that can aid you in program development and refinement. Included in this part are DOS tools (such as STAMP, MAKLABEL, and GDIR), the font editor, the on-line tags facility, the PenPoint Bitmap editor (which you use to create icons), and the PenPoint screen shot utility.

Master Index contains all the index listings for *PenPoint Application Writing Guide*, *PenPoint Development Tools*, *UI Design Reference*, and the *PenPoint Architectural Reference* volumes. It does not include index entries for the *PenPoint API Reference.*

PenPoint UI Design Reference 2.4

The *PenPoint UI Design Reference* has two parts:

PenPoint UI Specification describes the PenPoint Notebook User Interface (NUI) as it is implemented in PenPoint.

PenPoint UI Design Guidelines suggests how you should use the PenPoint NUI when designing your applications.

PenPoint Architectural Reference 2.5

This is the bible of all the published software interfaces to PenPoint. It is divided into several parts. Each Part talks about some subsystem in PenPoint, listing its features, explaining its concepts, describing its class hierarchy, and discussing how to use it.

The following sections describe the parts in the PenPoint Architectural Reference and attempts to summarize the features of the classes that each part documents.

Class Manager 2.5.1

Part 1: Class Manager describes the PenPoint class manager.

The Class Manager is a subsystem that supports object-oriented programming in PenPoint. The Class Manager provides the message-passing capability in PenPoint, and it provides the mechanisms for object creation and class inheritance. The PenPoint Class Manager and the built-in classes, together with the SDK

Development Tools, comprise a complete object-oriented software development environment for the rapid prototyping and delivery of finished applications in PenPoint.

The PenPoint Class Manager supports these features:

Message handling. The Class Manager provides a set of C functions and macros for message handling. These are standard C constructs; there is no special language support for message passing as in C++ or Smalltalk.

Object Creation. To create an object, you send the message **msgNew** to a class. The Class Manager does the rest. To create a new class you send the message **msgNew** to a special class, **clsClass**. To copy an object, you send the message **msgCopy** to the object to copy.

Class Inheritance. New classes can inherit functionality from ancestor classes.

Root Classes. The Class Manager defines two root classes, **clsObject** and **clsClass**, from which all other classes are derived.

Message types. Macros are provided for building message tokens. Tokens are differentiated according to the classes that defined them to provide a separate name space for each class.

Synchronous and asynchronous message passing. Because PenPoint is a multi-tasking operating system, you send messages to other tasks. If you want to wait for a response from a message that you send to another task, you can use synchronous message passing. If you don't want to wait for a response, you can use asynchronous message passing.

Single occurrence of executable code. Executable code that supplies an object's behavior is not duplicated in the class hierarchy. If the behavior is inherited from an ancestor class, the code for that behavior remains in the ancestor class.

Observing objects. You can set up an object so that it has observers. Observers are other objects that are interested in being notified of particular state changes in the observed object. Objects that want to become observers of a target object need only send the target a message requesting observer status.

Unique Identifiers. Each object in PenPoint is identified with a unique identifier (UID). These identifiers can be constant (well-known) or dynamic. You use well-known UIDs to identify classes and other objects that must be known to any client that wants to use them. In order to make well-known UIDs unique, a portion of the identifier value is administered by GO Corporation. Contact GO Developer Support to obtain unique UID numbers.

Application Framework 2.5.2

Part 2: Application Framework describes the PenPoint Application Framework, which provides you the tools you use to allow your application to run under the notebook metaphor. It provides mechanisms for:

- Installing applications on PenPoint (used by the Installer).
- Creating documents.
- Activating documents (turning to a page in the notebook).
- Saving and restoring document data.
- Deactivating and deleting documents (turning away from the page).
- De-installing applications.
- Embedding documents in other documents, which is the foundation for the PenPoint Embedded Document Architecture (EDA).

Windows and Graphics 2.5.3

Part 3: Windows and Graphics describes ImagePoint, the imaging system for the PenPoint operating system, and how applications can control the screen (or other output devices).

The window system supports:

- Multiple overlapping windows in a window tree.
- Windows on any windowing device; windowing devices include the screen, printers, and memory.
- Many windows. Hundreds can be on-screen at once.
- Control over window clipping, visibility, and transparency.
- Notification when windows are dirty and need to repaint.
- A window layout protocol which lets windows dynamically size themselves and their children.

The ImagePoint imaging model supports:

- Device-independence.
- Arbitrary coordinate system with client-specified:
    - units
    - scaling
    - rotation
    - translation
- RGB color specifications.
- Fill and stroke painting model.
- Polylines, rectangles, ellipses, Bezier curves, arcs, chords.
- Control over line thickness, ends, and corners.

- Gray-scale sampled images ("bitmaps"), including dithering and scaling.
- Outline fonts with these additional features:
  - Hints for rendering fonts in small sizes.
  - Missing fonts are synthesized from closest matching font.
  - Multiple font encodings supported.
  - Substitution for missing characters in the current font.
  - Font editor (supplied as part of the SDK).

UI Toolkit 2.5.4

Part 4: UI Toolkit describes the PenPoint classes that implement many of the common features required by the PenPoint user interface. These features include:

- Borders, which support some common features of toolkit window repainting, such as margins, and background and foreground colors.
- General layout classes, which implement approaches for positioning and sizing child windows.
- Controls, which implement the translation of window input messages into messages sent to self and other objects.
- Labels, which are a simple way of displaying small amounts of text.
- Buttons, which are labels that the user activates.
- Tookit tables, which support the initialization, layout and notification management for groups of toolkit components (such as buttons). The capability to organize components in a group is used by several subclasses.
- Menus and menu buttons. Menus are special toolkit tables that often group several buttons and menu buttons; menu buttons are special buttons that display a menu.
- Scroll bars, a special descendant of borders that handles a lot of the work of scrolling for you.
- List boxes, which are scrolling windows that support very large numbers of entries. Unlike a table, only those entries currently visible in the list box need have a window. Descendants of list boxes provide scrolling lists specifically for strings and for font names.
- Text fields, which are labels that the user can write upon. Subclasses of text fields have additional semantics to support integer fields, date fields, fixed-point fields, and so on.
- Notes, which present transient information to the user. PenPoint also provides standard message functions that display information using notes.
- Frames, which manage a collection of other UI components and a client window. Most applications use frames for their main windows, dialog boxes, and pop-up windows. Frames can have decorations, such as close boxes, title bars, tab bars, and command bars.

- Option sheets, which present the user with choices for attributes or settings. Because the user can leave an option sheet on-screen, the interaction between option sheets, option cards, and the selection is necessarily quite complex.
- Icons, which show an iconic menu button made up of a bitmap image and a string.
- Trackers, which grab input and draw transient "rubber-banding" figures in response to pen movements. The toolkit uses this to provide feedback when the user drags or resizes items.
- Progress bars, which graphically display the relationship between two values (usually one value represents a total while the second value represents percentage of the total).

Input and Handwriting 2.5.5

Part 5: Input and Handwriting describes the PenPoint input system and programmatic access to the handwriting translation subsystems. The input system provides:

- Programmatic control of input filtering, including priority of filters.
- Capability to receive all input events not intercepted by the filters (grabbing).
- Routing of input events to the appropriate object (listeners).

The handwriting translation subsystem provides:

- Handling of accumulated pen input (strokes).
- Shape recognition.
- Gesture interpretation.
- Handwriting translation.

Text Component 2.5.6

Part 6: Text Component describes the PenPoint facilities that allow any application to provide text editing and formatting capabilities to its users. The text component allows:

- Your code to display both plain and fancy text to the user in one or more text data objects.
- The user to interact with the text to modify both the characters and their appearance.
- The user to transfer all or part of the text from one text data object to another (possibly non-text) object, and vice versa.
- Your code to file text data objects.
- Your code to observe and direct the user's interactions with the text.

- Embedded objects, which are used to implement insertion pads and signature pads, and can include graphics, spread sheets and other applications in documents.

There is a difference between displaying text through the graphics subsystem and using the Text component. You can use the graphics subsystem to display characters on the screen, but users can't dynamically manipulate the text. Furthermore, the text component includes paragraph and document attributes that define things such as margins and tabs.

File System 2.5.7

Part 7: File System describes the PenPoint File System. The File System allows you to:

- Create, open, close, and delete files.
- Read and write file data.
- Copy, rename, and move files and directories.
- Seek to a new position within a file or find out the current byte position within a file.
- Modify file and directory attributes.
- Create user-defined attributes for files and directories.

System Services 2.5.8

Part 8: System Services describes the function calls that applications can use to access kernel functions, such as memory allocation, timer services, process control, and so on.

PenPoint provides basic OS-level services in areas such as task and memory management, as well as run-time language extensions. This part of the *PenPoint Architectural Reference* explains the use of these services. The PenPoint kernel and run-time libraries provide a level of functionality to the applications programmer collectively called **System Services**.

The kernel interface includes task management functions, memory management functions, and inter-task communication functions; these are the low-level PenPoint controls for the Intel multi-tasking environment.

The run-time libraries include functions for string manipulation, character handling, port I/O, buffer manipulation, memory allocation, and time stamp operations. There is a separate library for fixed-point arithmetic functions, which includes both error-checking and fast non-error-checking routines.

Note that there are no file-handling functions in System Services. The PenPoint File System has its own object-oriented API, described in *Part 7: File System* of the *PenPoint Architectural Reference*. The C run-time library implements most of the C standard I/O functions such as `fopen` and `fread` on top of this object-oriented API.

Utility Classes 2.5.9

Part 9: Utility Classes describes a wide variety of classes that save application writers from implementing fundamental things such as list manipulation, busy clock handling, and so on.

- **clsList** provides a fundamental set of tools for creating and managing a list of 32-bit values. It is no coincidence that UIDs and pointers are also 32-bits long. You can use these objects to store lists of UIDs or pointers to larger structures and you can pass these list objects to other objects.
- **clsStream** provides the basic messages used to communicate with a stream device. Many other classes descend from **clsStream**, such as **clsFileSystem** (the File System) and **clsSio** (the Serial Port class).
- The browser allows you to create a browser window or a table of contents on screen so that the user can manipulate the files and directories or documents and sections.
- File import and export uses messages from the browser to import files as PenPoint documents and to export PenPoint documents as files.
- The Selection Manager provides an central manager that keeps track of the selection owner. The selection manager notifies observers when the selection changes.
- The Transfer Class provides the messages and functions that implement the PenPoint operating system transfer protocol, which objects can use to exchange data.
- The Quick Help API provides a simple way to provide help to users. When the user makes a question mark gesture on a window, the quick help manager locates the quick help resources associated with that window and displays the resources on screen.
- The Busy Manager allows applications to inform the user when a time-consuming operation is taking place, thereby reassuring the user that the machine is still running.
- The search and replace API provides the protocol and traversal driver to search and replace text strings in embedded objects.
- The Undo Manager enables applications to respond to the Undo command to undo user interface actions.
- **clsByteBuf** and **clsString** implement simple data objects which file byte arrays and null-terminated strings.
- **clsTable** provides a general-purpose table component using a row and column metaphor to implement random and sequential access to data in a file.
- **clsNotePaper** provides a user interface to the PenPoint operating system's note-taking building block. Using this building block, your application can provide users with a way of managing ink on screen.

Connectivity 2.5.10

Part 10: Connectivity describes the classes that applications can use to access remote devices. The PenPoint remote interface includes:

- A consistent, object-based interface to ports and devices.
- A service architecture that enables users to install drivers and other non-application programs without rebooting or otherwise interrupting work.
- Support for serial and parallel ports.
- Support for fax and data modems.
- Deferred data transfer through an In box and an Out box.
- Access to networks.
- An address book protocol.

Resources 2.5.11

Part 11: Resources describes how to read, write, and create PenPoint resource files. Resources include the following features:

- Ability to store data and object resources in files which can be replaced by the user.
- Agents that can read and write resources that use unique data formats.
- GO provides agents to handle standard data formats.
- Applications can access a set of resources through predefined resource file lists.
- You can define data resources in a C file and compile the file to a resource file.

Installation API 2.5.12

Part 12: Installation API describes PenPoint support for installing applications, services, fonts, dictionaries, handwriting prototypes and so on.

PenPoint provides an installer through which users install and configure applications, fonts, services, dictionaries, and so on. This frees developers from having to write their own installer. The system-provided installer also frees users from having to learn a different installer for each product that they purchase; they just use the PenPoint installer.

The PenPoint Installer uses several system concepts and components:

- All distribution volumes for PenPoint software have the same file organization.
- Dynamic Link Libraries and control files allow products to specify the components that they require. The PenPoint Installer can then load the components only if they are not present in the system already. The Installer can also unload components when they are no longer needed.

- The installation manager class handles installation. If a product requires special treatment not provided by existing installation managers, you can subclass the installation manager class.
- Application monitors exist for each installed application. The application monitors assist in application installation and deinstallation.

Writing PenPoint Services 2.5.13

Part 13: Writing PenPoint Services describes how to write a PenPoint service.

The PenPoint Services Architecture provides a framework for separately installable, non-application DLLs that provide non-application system extensions to applications, components, and other services. The most common use for services is as MIL services (device drivers), but services can also be used for database engines, E-mail backends, and so on.

Services provide these features:

- Targeting other services to form chains of services.
- Delayed binding to targets (the target doesn't have to exist when the service is created.
- Ownership of services and restricted access to services, based on ownership.
- Notification of connection and disconnection of devices and services.
- A service manager that manages installed services and controls client access to services.

API Reference 2.6

The *API Reference* provides datasheets for each function and message in each subsystem described in the *Architectural Reference*. These are generated from the header files in \PENPOINT\SDK\INC.

The parts in the *PenPoint API Reference* are organized identically to the parts in the *PenPoint Architectural Reference*. However, within each part, the classes are described in no particular order, other than the relative importance to the central topic of the part. If you are looking for a particular class or message, it is a good idea to use the index.

This volume (*PenPoint Development Tools*) contains a Master Index to all manuals in the PenPoint SDK documentation set.

Contents of the SDK

2.7

Table 2-2 lists the directories and key files in the PenPoint SDK and describes their contents.

Table 2-2
SDK Contents

Directory or File	Comments
	PenPoint
\PENPOINT\APP	PenPoint applications.
\PENPOINT\BOOT	Control files, start-up code.
\PENPOINT\BOOT\APP	PenPoint system applications (Notebook, Bookshelf, etc.).
\PENPOINT\BOOT_APP	Optional DEBUG versions of system apps.
\PENPOINT\BOOT\DLL	System DLLs.
\PENPOINT\BOOT_DLL	Optional DEBUG versions of system DLLs.
\PENPOINT\BOOT*.INI	Control files.
\PENPOINT\BOOT*.DLC	Control files of DLLs loaded at start-up.
\PENPOINT\BOOT_*.*	Backup copies of control files.
\PENPOINT\BOOT\MIL.OS	Machine Interface Layer code.
\PENPOINT\BOOT\PENPOINT.OS	PenPoint kernel.
\PENPOINT\BOOT_PP.OS	DEBUG version of PenPoint kernel.
\PENPOINT\BOOT\PPBOOT.EXE	DOS program that loads PenPoint.
\PENPOINT\FONT	PenPoint fonts.
\PENPOINT\HWX	Handwriting prototypes.
\PENPOINT\PDICT	Home of personal dictionary.
\PENPOINT\PREFS	home for preference settings.
\PENPOINT\SERVICE	System services used by PenPoint.
\PENPOINT_SERVICE	Optional DEBUG versions of system services.
\PENPOINT\SDK	
\PENPOINT\SDK\APP	PenPoint applications that are not loaded in the default PenPoint configuration; some of these are SDK tools such as the bitmap editor and S-Shot tool, others are applications that you must distribute with your own application if it requires them.
\PENPOINT\SDK_APP	Optional DEBUG versions of non-system apps.
\PENPOINT\SDK\DLL	PenPoint DLLs that are not loaded in the default PenPoint configuration that you must distribute with your own application if it requires them.
\PENPOINT\SDK\INC	PenPoint header files.
\PENPOINT\SDK\INC\SYS	Some header files are in this subdirectory for ANSI C compatibility.
\PENPOINT\SDK\LIB	library information to enable you to link your code with routines and externals in PenPoint DLLs and PENPOINT.OS.
\PENPOINT\SDK\SAMPLE	Sample application code and sample Watcom WMAKE Makefiles.
\PENPOINT\SDK\UTIL\CLSMGR	PenPoint Method Table compiler.
\PENPOINT\SDK\UTIL\DOS	DOS utilities for PenPoint.

continued

Table 2-2 (continued)

Directory	Comments
	Compiler Tools
\WATCOM\BIN	DOS-only Watcom compiler tools.
\WATCOM\BINB	Dual mode (DOS and OS/2) Watcom compiler tools.
	Optional Goodies
\GOODIES\SDK\APP	Unsupported applications.
\GOODIES\SDK\BOOT\DLL	Unsupported developer DLLs.
\GOODIES\SDK\UTIL	Various unsupported developer utilities.

The files on the Goodies disk are unsupported, however you might find them useful. Most software on the Goodies disk has a README.TXT file in its subdirectory that explains its use. To reiterate: software on the Goodies disk is unsupported. PenPoint Developer Support will not answer questions related to anything on the Goodies disk.

Chapter 3 / Running PenPoint on a PC

Although the PenPoint™ operating system is targeted for mobile pen-based computers, it is possible to run PenPoint on *some* PC configurations. The simulation is imperfect (no static RAM, no pen-on-screen interaction, and so on) but it is useful for development and debugging. The mouse simulation of a pen does not have the same user interface as a PenPoint computer at all. Using a pen tablet attached to a PC lets you hand write more naturally, but you still aren't able to directly touch objects on-screen or write on what you see.

PenPoint will not work on all PCs, even those advertised as "100% IBM PC compatible," *even those* claiming compatibility with the requirements below.

Hardware 3.1

To run PenPoint on a PC, you need a 386-class machine with a hard disk that has at least 25 Megabytes free

If you want to run PenPoint on your PC, in addition it must have the following:

- 8 MB of RAM configurable as extended memory. You may be able to develop with less memory, however the PenPoint Source-level Debugger with full symbol tables and all applications loaded takes around 6 MB.
- IBM VGA or Compaq VGA video adapter (because PenPoint has its own video drivers, adapters advertised as being 100% compatible with these under DOS may not work).
- Either a mouse or a digitizing tablet.

To repeat, your PC may meet these requirements yet not be able to run PenPoint. It may even be able to run one release, yet not another. Table 3-1 lists the computers that GO has tested and supports for PenPoint development.

Table 3-1
Tested Machine Configurations

Machine	Configuration
Compaq 386/20E	20MHz 386, no coprocessor, Compaq BIOS dated 31-May-89, DOS 5.0, 640KB + 8192 Extended = 8,832KB total memory.
Dell 325P	25MHz 386, no coprocessor, Phoenix BIOS dated 27-Sep-91 version 1.10A16, DOS 5.0, 640KB + 7,160 extended = 7,800KB total memory.
IBM PS2/70	25MHz 386, no coprocessor, IBM BIOS dated 2-Feb-89, DOS 5.0, 640KB + 7,424 = 8,064KB total memory.
IBM PS2/70	IBM BIOS dated 11-Apr-88, 640KB + 5,376 = 5,980KB total memory.
IBM PS2/80	
IBM PS2/90	33MHz 486, built-in coprocessor, IBM BIOS dated 30-Jan-91, DOS 5.0, 640KB + 11,520 extended = 12,160KB total memory.
IBM PS2/95	25MHz 486, built-in coprocessor, IBM BIOS dated 8-Jan-90, DOS 5.0, 640KB + 7,424 extended = 8,064KB total memory.
NCR 386sx/MC20	20MHz 386, no coprocessor, NCR BIOS version 1.01.00 and Phoenix BIOS dated 15-Oct-90 version 1.02.07, DOS 3.30, 640KB + 7,424 extended = 8,064KB total memory.

GO has also tested PenPoint on the machines listed in Table 3-2, but makes no commitment for continued support.

Table 3-2
Machine Configurations with No Committed Support

Machine	Configuration
Northgate 386 33E	Model FCH, 33MHz 386, no coprocessor, Northgate (AMI) BIOS dated 15-Sep-89 version 3.2B, DOS 4.01, Video 7 VGA version 1.09, 40MB type 22 drive, 640 + 7,552 extended = 8,192KB total memory.
Samsung S800	20 MHz 386, no coprocessor, AMI BIOS version 3.04 03B dated 30-Nov-87, DOS 5.0, ATI Basic 16 VGA, type 34 Western Digital IDE controller, Conner 201 MB drive.
Toshiba 5200/100	20 MHz 386, no coprocessor, Phoenix BIOS dated 15-Jan-88, DOS 4.01, built-in VGA, 640 + 7,168 extended = 7,808KB total memory, 102MB drive.
Uniq 486/33	33 MHZ 486, built-in coprocessor, American Megatrends, Inc. (AMI) 486 BIOS for Vantage 486 version 2.0 dated 7-Jul-91, DOS 5.0, ATI VGA Wonder XL VGA w/ 1MB, 640 + 7,424 extended = 8,064KB total memory, type 47 controller.

Mouse 3.1.1

If you do not have a digitizing tablet, you can use a Logitech C7 or C9 serial mouse, a Microsoft compatible mouse, a PS/2 mouse, or a bus mouse. GO does not currently support the Logitech Mouseman or newer Logitech mice.

You may need to load the DOS driver for your mouse before starting PenPoint; either specify the device driver MOUSE.SYS in your CONFIG.SYS file or make sure to run MOUSE.COM before you start PenPoint.

Memory, Caches, and RAM Disks

3.1.2

You need 8MB of RAM to run PenPoint. It must be extended memory, not expanded memory. Memory-resident software may use up much of your memory. PenPoint might run in less memory, but our QA process has only evaluated 8 MB machines.

The Watcom compiler and linker (WCC386P and WLINKP) are protected-mode programs that use extended memory. If you use large disk caches or RAM drives, they may not have enough memory to run. Also, the DOS 5 EMM386.EXE driver conflicts with their use of memory. For more information, see the section on compiling.

PenPoint will not work if you run a disk cache such as SmartDrive. It also interfers with some RAM disks. We believe that VDISK can coexist with PenPoint, but others may not.

You can check whether your RAM disk is compatible through these steps:

- Run PenPoint (PPBOOT.EXE).
- Exit back to DOS.
- If the size or contents of your RAM disk has been altered, assume that it and PenPoint conflict.

Networks

3.1.3

To stop running PenPoint on a PC, you either tap the shutdown button in the Preferences Power option sheet or press Pause and enter **q** at the prompt. Usually your computer returns to DOS. However, if your computer has loaded network software, you may have to reboot your computer after running PenPoint (because the machine will hang). If you want to avoid this reboot, do not load your network software before you run PenPoint or remove it from memory. In either case, GO recommends that you run CHKDSK to verify the state of the DOS file system.

Labeling Volumes

3.1.4

Warning You can use PenPoint to create a disk label in lower case letters. Such a label cannot be changed by LABEL in DOS 5.0.

Unlike the DOS convention of listing disks by their drive letter (C: for example), PenPoint refers to disks by their volume names (not by their devices). You can name volumes when you format them, or with the DOS LABEL command. It's a good idea to give volume names to all the floppies and hard drives you will be using with PenPoint.

PenPoint volumes are indicated by a \\ before the volume name. Thus \\MYDISK\DIR\FILE1 is a path on a volume labelled MYDISK.

There are a few reserved volume names in PenPoint. \\RAM is reserved for the PenPoint memory file system, which is usually in static RAM, but can be on-disk. \\BOOT is the volume from which PenPoint booted (programmatically known as **theBootVolume**). When running PenPoint on a PC, it's safe to refer to files relative to \\BOOT; however, when booting a PenPoint computer from floppies, your configuration information must refer to the actual floppy disk volume labels.

If PenPoint encounters a volume name in a path that it does not recognize, it will prompt the user to insert the volume with that name.

Setting Up Initialization Files 3.2

When PenPoint starts, it reads several initialization files in \PENPOINT\BOOT. These set certain environment variables and tell it what dynamic link libraries (DLLs) to load, and what applications to install. Before running PenPoint for the first time, you will probably need to modify at least one of these files.

All files you need to run PenPoint on a PC are in the \PENPOINT directory that you installed with your SDK. The document on installing PenPoint shipped with your software contains the latest information on installing PenPoint and machine requirements to run it.

Default versions of the initialization files necessary to run PenPoint on a PC are in \PENPOINT\BOOT. The SDK contains two versions of each initialization file. One is preceded by an underscore (_), the other isn't. You should modify the file that does not contain the underscore, and leave the one with the underscore so that you can refer to it later.

If, after changing the originals, you have difficulty running PenPoint, compare the modified file with its underscored equivalent.

Two notes on initialization files:

- The last character in your initialization files must be a newline (carriage return, line feed). If the newline is not present, PenPoint will ignore the last line in your file.
- If the line is of the form `tag=value` and you have two lines that define the same tag, PenPoint uses the first occurence and ignores all subsequent occurrences of the tag.

Table 3-3
PenPoint Initialization Files in \PENPOINT\BOOT

Name	Description
MIL.INI	Hardware interface settings for PC MIL.
ENVIRON.INI	PenPoint configuration settings.
BOOT.DLC	List of system DLLs required to run PenPoint.
CONSOLE.DLC	A subset of BOOT.DLC for single-screen debugging.
SYSCOPY.INI	List of files and directories to copy to memory.
SYSAPP.INI	System applications to install.
APP.INI	Other applications to install.

PenPoint Boot Sequence 3.3

When you run PPBOOT.EXE on a PC, it locates the MIL.INI file and uses the information in MIL.INI to configure the hardware, initialize the MIL (machine interface layer), and start the PenPoint operating system.

PPBOOT determines the volume from which PenPoint is being booted, and assigns the boot volume to the symbol **theBootVolume**. You can use **theBootVolume** in your code to refer to this volume.

The operating system locates the ENVIRON.INI file and uses that information to configure things such as the size of the display, the selected volume, the time zone, and the name of the first application to run. If a \PENPOINT directory does not exist on the volume, PenPoint creates one.

The PenPoint operating system then looks for \PENPOINT\BOOT\BOOT.DLC and it initializes each DLL listed. PenPoint then copies all the files and directories listed in \PENPOINT\BOOT\SYSCOPY.INI to the selected volume.

PenPoint installs the system applications listed in \PENPOINT\BOOT\SYSAPP.INI. The last DLL in BOOT.DLC, AUXNBMGR.DLL, installs the applications listed in \PENPOINT\BOOT\APP.INI.

MIL.INI 3.4

The file \PENPOINT\BOOT\MIL.INI describes configuration of the PC on which the PenPoint operating system will run. Although most devices are installable, there are a handful of devices that PenPoint must know about ahead of time, such as the screen type, the stylus device, and the system clock.

MIL.INI is specific to the implementation of the PenPoint MIL (machine interface layer) for a PC. Because PC configurations can vary so much, the PC MIL implementation reads the configuration information from MIL.INI. The MILs that run on most tablet hardware will be preconfigured for a specific machine and thus will not need to read configuration information from a file.

Important MIL.INI is specific to the implementation of the PC MIL. Other MILs probably won't have a MIL.INI file.

You *must modify* \PENPOINT\BOOT\MIL.INI before you can boot. At a minimum, you need to specify your pointing device and monitor in MIL.INI. Another common change is to specify the port (if any) on which debug output should appear. MIL.INI contains instructions for how to modify itself together with commented-out versions of common settings.

Changing MIL.INI does not change an active system. Changes take effect the next time you boot PenPoint.

The format is of a setting in MIL.INI is:

keyword = *value*

Following is a list of the _MIL.INI file shipped with the SDK:

```
# The following applies to all serial pointing devices:
#
# Use the UNIPENPORT tag to set the serial port. For example:
#    UNIPENPORT = 1          - This will set unipen to COM1:
#    UNIPENPORT = 760,5      - This will set unipen to comport at 2F8 irq 5
#
# Use the UNIPENTYPE tag to select a predefined protocol:
#    UNIPENTYPE = WACOM510C  - This sets unipen to the Wacom 510C digitizer
#
# MICROSOFT:    Microsoft serial, two-button mouse
#
# LOGITECH:     Logitech C7 or C9 serial, three button mice. For a Logitech
#               MouseMan use the MICROSOFT tag
#
# WACOM510C:    The switches must be set as follows (X = ON, O = OFF) -
#
#                        DS1      DS2      DS3
#               Front  OXOXXOOX XXOXOXXO XXOOXXOO  Back
#
# WACOM510:     The older Wacom units, red power LED, attached power cord.
#               The switches must be set thusly (X = ON, O = OFF) -
#
#               Front  OOOXOOXO XXOXOXXO XOOOXOOO OOOOOOOO  Back
#
# SuperScriptII: The SuperScript II LCD/digitizer combo. For the ink to
#               be aligned you must specify "ScreenType=SuperScript" or
#               "ScreenPixelsPerMeter=3690" in MIL.INI.
#
# CalCompDBII: CalComp's DrawingBoard II, should also work with their
#               "Wiz" product. Use the default "Hi Resolution Binary"
#               format (#23) in "run" mode, 9600 baud no parity, eight
#               data bits, one stop bit, 1000 lpi resolution, 125 pps.
#               Enable "Send when out of proximity", button 10=1 in bank B.
#
# The following are generic descriptions. You will probable have to "tune"
# these by placing the parameters into MIL.INI and adjusting for
# the specific charatersitics of your digitizer.
#
# GAZELLE:      For products from Gazelle System (now owned by Logitech)
# MM:           The common "MM" digitizer protocol.
# BITPAD2:      BitPad 2 binary protocol
# BITPAD2ASC:   BitPad 2 ACSII protocol
#
# See the universal pen driver manual for details on how to define a
# custom serial protocol.
#
# Remove the "#" on the line below for the port of your pointing device:
#UNIPENPORT = 1
#UNIPENPORT = 2
#
# Remove the "#" on the line below which matches your serial pointing device:
#
#UNIPENTYPE = MICROSOFT
#UNIPENTYPE = LOGITECH
#UNIPENTYPE = CalCompDBII
#UNIPENTYPE = GAZELLE
#UNIPENTYPE = SuperScriptII
#UNIPENTYPE = WACOM510C
#UNIPENTYPE = WACOM510
#UNIPENTYPE = MM
#UNIPENTYPE = BITPAD2
#UNIPENTYPE = BITPAD2ASC
```

```
# This supports a Microsoft "bus" or "InPort" mouse.
# Remove the "#" on the next line if your bus mouse is the primary one
#BusMousePort=Primary
# Remove the "#" on the next line if your bus mouse is the secondary one
#BusMousePort=Secondary
# Remove the "#" on the next line to enable the PS/2 mouse port.
# Note: do not confuse a PS/2 mouse port with a Microsoft bus mouse port.
#PS2Mouse=on

# Remove the "#" from "LeftyMouse" if you are left-handed and have a
# two button mouse. This will swap the meaning of the buttons so that
# the right button will mean "tip-down" and the left button "out of prox".
#LeftyMouse=true
# The following lines are concerned with the Wacom "310" coprocessor. They
# can be used to describe the alignment and rotation of the pen sensor
# relative to the LCD panel. The X axis is the long axis, the Y the short.
# All numbers are in decimal (base 10). The defaults for the GO "Hyde"
# prototype are listed:
# Remove the "#" to enable the Wacom 310 MIL device:
#Wacom310=on
# This tag is used to set the address of the command I/O port
#Wacom310Cmd=1222
# This tag is used to set the address of the data I/O port
#Wacom310Data=1220
# This tag is used to set the Wacom 310 interrupt level
#Wacom310Int=10
# These two tags can be used to align the pen in the case of a constant error.
# Signed values are supported. The units are 0.1 millimeters
#Wacom310XOffset=0
#Wacom310YOffset=0
# These two tags describe the maximum values for the X and Y axis
#Wacom310MaxXPos=2240
#Wacom310MaxYPos=1400
# These two tags may be use to independently flip either axis. When used
# together the pen coordinates can be rotated 180 degrees.
#Wacom310InvertX=yes
#Wacom310InvertY=yes
# Remove the "#" on the next line to route debugging information to COM1
#SerialDebugPort=1
# Remove the "#" on the next line to route debugging information to COM2
#SerialDebugPort=2
# Remove the "#" from "MonoDebug" ONLY if you don't want debugging info
# to be sent to an existing monochrome video card (the "second head")
#MonoDebug=off
# Remove the "#" on the line below which matches where your wish to route
# the lowlevel debugging information. If no "LowLevelDebug" is specified then
# these low-level messages will be supressed
#LowLevelDebug=mono
#LowLevelDebug=com1
#LowLevelDebug=com2
# Remove the "#" from "Floppy=hardware" if you wish to enable the PC floppy
# driver
#Floppy=hardware

# Change "off" to "on" in this line to enable auto-polling of floppy drives
# when "floppy=hardware"
FloppyAttachment=off
# Remove the "#" from "Harddisk=hardware" if you wish to enable the PC hard
# disk driver which talks directly to the hardware
#HardDisk=hardware
```

```
# Set Com1 to true to enable Com1 Asynchronous SIO, false otherwise
Com1=false
# Set Com2 to true to enable Com2 Asynchronous SIO, false otherwise
Com2=false
# Set Lpt1 to false to disable Lpt1 high speed packet parallel port I/O
Lpt1=true
# Set Lpt2 to false to disable Lpt2 high speed packet parallel port I/O
Lpt2=true
# Remove the "#" on the line below which matches your screen and video board
# If no "ScreenType" is specified then the MIL defaults to STD480 (a standard
# VGA screen in landscape). An automatic "int10 18" is done by the MIL for
# STD480 to set the VGA card into mode 18 (12 hex): 640 by 480 graphics.
ScreenType=STD480
#ScreenType=EGANul
#ScreenType=NCR
#ScreenType=IBM
#ScreenType=Samsung
#ScreenType=Ymh400
#ScreenType=Hyde
#ScreenType=SuperScriptII
#ScreenType=ATI768
# Each of the above screen types defines a prefered initial orientation. If
# you wish to override this then read on:
#
# Initial screen orientation: for a conventional VGA screen these are:
#
#
#             North
#               -
#        |             |
#        |             |
#        |             |
#        |             |
#        |             |
#        |             |
#  West  |             | East
#        |             |
# (top   |             |  (bottom
#  in    |             |   in
#  char. |             |   character
#  mode) |             |   mode)
#        |             |
#        |             |
#        |             |
#               -
#        ^    South    ^
#        ^             ^
#        ^             last VGA scan line
#        ^
#        first VGA scan line
#
# Remove the "#" on the line below which matches your desired orientation:
#
#InitialScreenTop=North
#InitialScreenTop=South
#InitialScreenTop=East
#InitialScreenTop=West
# If you have a VGA display on your computer but the machine is not listed
# above then you may explicitly set the pixel spacing by completing the
# line below and removing the "#". Typical values range from 2500 to 3700:
#ScreenPixelsPerMeter=????
```

```
# Remove the "#" on the line below which matches the your flash card type.
#FLASHCARD=TOSHIBA
#FLASHCARD=PC

# Remove the "#" on the line below if you are having trouble exiting
# from PenPoint back to DOS
#ExitToDOS=reset
```

Capitalization in the settings isn't significant. This table lists the MIL.INI keywords used by PenPoint.

Table 3-4
MIL.INI Keywords

Keyword	Meaning
	Serial Pointing Devices
UniPenPort	Sets the serial port for universal pen device.
UniPenType	Sets a predefined universal pen protocol. See the complete discussion of the universal pen driver later in this chapter for details on how to define a custom serial protocol.
BusMousePort	Specifies whether a bus mouse is secondary to another mouse device.
PS2Mouse	Enables the PS/2 mouse port.
LeftyMouse	Switches buttons on two or three button mouse for left-handed users.
	Wacom 310 Digitizer
Wacom310	Enables a Wacom 310 coprocessor.
Wacom310Cmd	Sets the address of the command I/O port.
Wacom310Data	Sets the address of the data I/O port.
Wacom310Int	Sets the Wacom 310 interrupt level.
Wacom310XOffset	Sets a constant X offset.
Wacom310YOffset	Sets a constant Y offset.
Wacom310MaxXPos	Specifies the maximum X position.
Wacom310MaxYPos	Specifies the maximum Y position.
Wacom310InvertX	Inverts the X axis.
Wacom310InvertY	Inverts the Y axis.
	Debugging Information
SerialDebugPort	Specifies a serial port on which to write debugging information.
MonoDebug	Turns off debugging information for a second (monochrome) monitor (when off).
LowLevelDebug	Specifies destination for MIL debugging information.
	Disks
Floppy	Enables floppy disks through hardware or BIOS.
FloppyAttachment	Enables autopolling of the floppy drive to detect attachment.
HardDisk	Enables hard disks accessed through hardware or BIOS.
	Asynchronous Serial I/O
Com1	Enables COM1 asynchronous serial I/O.
Com2	Enables COM2 asynchronous serial I/O.

continued

Table 3-4 (continued)

Keyword	Meaning
	High-Speed Packet Parallel Port I/O
Lpt1	Enables LPT1 high-speed packet parallel port I/O.
Lpt2	Enables LPT2 high-speed packet parallel port I/O.
	Video Controller
ScreenType	Specifies the screen type.
InitialScreenTop	Specifies the screen top edge at startup.
ScreenPixelsPerMeter	Specifies the pixel spacing of a non-standard VGA screen.
	TOPS FlashCard Type
FlashCard	Specifies TOPS FlashCard type (if necessary).
	Exit to DOS
ExitToDOS	Some PCs have trouble exiting PenPoint and returning to DOS. If you notice this problem, specify "reset."

In general you enable a device that uses a serial port by assigning it a port. If no assignment is made then the device will disable itself. Because of this scheme, there are no default serial ports.

You need not and should not enable `Com1` or `Com2` to use a serial port for mouse input or debug output. `Com1` and `Com2` tell PenPoint what serial ports are available for use by PenPoint services, not MIL devices.

ENVIRON.INI File 3.5

The file \PENPOINT\BOOT\ENVIRON.INI contains a set of environment settings for PenPoint. PenPoint system software and other programs defines names and values in ENVIRON.INI. Some of the DLL's use this to get initialization information such as the type of display, the monitor orientation, the default volume, and so on.

You should modify \PENPOINT\BOOT\ENVIRON.INI to enable logging by uncommenting the line

```
#DebugSet=/D*1 /DD8000
```

This sets up logging to the file \PENPOINT.LOG on the boot volume.

This is probably the only change you need to make to the default ENVIRON.INI in order to run PenPoint; the default settings for the variables described below are correct.

Like MIL.INI, changing ENVIRON.INI does not change an active system. Changes take effect the next time you boot PenPoint.

The format is of a setting in ENVIRON.INI is:

keyword=*value*

This is a list of the _ENVIRON.INI file shipped with the SDK.

```
## PenPointPath=
##
## Initial Penpoint application to run
StartApp = \\boot\penpoint\boot\app\bookshelf

## Alternate screen metrics, in device units
#ScreenWidth=200
#ScreenHeight=320

## Size of the swap file.
## This is number of bytes in swap file, in hexadecimal; e.g. 5MB = 500000.
SwapFileSize=500000

## Boot configuration. Choices are: DebugRAM, DebugTablet and Tablet.
Config=DebugRAM

## Timezone setting used by ANSI time routines (see WATCOM library ref)
TZ=PST8PDT

## Initial setting of the debug flags
##  /D*1: Validates all heap allocates and frees under the debug penpoint.os.
DebugSet=/D*1
#DebugSet=/DD8000 /D*1

## Forces a flush after N or more characters are written, 1 flush per
## call to DebugBuf (called by Debugf, DPrintf).
#DebugLogFlushCount=0

## Version string used by Preferences
Version=PenPoint|386 (386.45H)|Copyright c 1992, GO Corporation|All Rights Reserved.

## Path to a default bookshelf, copied at boot time
#BkshelfPath=\\boot\penpoint\boot\doc

## Progress Bar Maximum
bootProgressMax=250

## Open the notebook at boot time
Autozoom=Notebook

## Root of PenPoint
```

Capitalization in the settings isn't significant. The following table lists the ENVIRON.INI keywords used by PenPoint.

Table 3-5
ENVIRON.INI Keywords

Keyword	Meaning
AutoZoom	Specifies the name of a document to open in the window after booting.
BkShelfPath	Specifies the path to the bookshelf directory.
BootProgressMax	Used in the booting progress indicator. *Do not modify.*
Config	Specifies the disk and debugging environment.
DebugLog	Specifies the name of a file for debugger output.
DebugLogFlushCount	Specifies the when to flush the debug log to a file.
DebugSet	Specifies one or more debug flags.
PenPointPath	Specifies the location of the PENPOINT directory.
PenProxTimeout	Specifies a delay before sending an out of proximity event.
ScreenHeight	Specifies the height of the screen in device units.
ScreenWidth	Specifies the width of the screen in device units.
StartApp	Specifies the first PenPoint application to run on startup.
StealMem	Specifies that PenPoint should use less memory.
SwapBoot	Specifies that PenPoint should load memory from a saved swap file.
SwapFileSize	Specifies the size of the swap file in bytes.
TZ	Specifies the time zone.
UndoLimit	Specifies the maximum number of undos.
Version	Specifies the PenPoint version label.
VolSel	Specifies the label of the volume used to create the PenPoint system.
WinMode	Specifies the initial orientation of the screen device.
ZoomMargin	Specifies the distance from the bottom of the autozoom document to the bottom of the screen.
ZoomResize	Specifies whether the autozoom document has a resize tab.

An application can add its own keywords to ENVIRON.INI for testing. To get a value from ENVIRON.INI, use the kernel function **OSEnvSearch**().

These sections describe the keywords in more detail:

AutoZoom 3.5.1

AutoZoom specifies the name of a document that PenPoint is to open on the desktop after booting. Usually this document is Notebook (meaning the main note book). However, you can specify any other document, provided its application is installed.

If you set ZoomMargin to 0 and specify ZoomResize as `FALSE`, the document that is opened with AutoZoom is locked into the page. If the document is a notebook, the user will not be able to close or resize the notebook. If the document is a page-based application, the user will not be able to turn away from the application.

BkShelfPath 3.5.2

BkShelfPath specifies the path to the bookshelf directory. Usually this contains \PENPOINT\BOOT\DOC.

BootProgressMax 3.5.3

BootProgressMax is used by the progress indicator that appears when you boot PenPoint. *Do not modify* this field.

Config 3.5.4

Config is used to specify in one place the selected volume, the location of installed code, and, in the future, debugging options. There are three environments that you can specify with Config:

DebugRAM Sets **theSelectedVolume** to RAM and stores installed code in RAM also. In this configuration, any changes you make when running PenPoint are not saved, but it is easier to clean up after and it makes it easy to install new versions of applications after each reboot.

Tablet Sets **theSelectedVolume** to be the drive on your PC from which you booted PenPoint and stores installed code on the disk. In this configuration, changes that you make are saved from one cold boot to the next. For other considerations on Config=Tablet, see the section Running in Tablet-Like Mode.

DebugTablet Currently is similar to Tablet. In the future, DebugTablet may be expanded to enhance debugging information in a tablet-like environment.

DebugLog 3.5.5

DebugLog specifies the name of a file to which debug stream data is directed. When you are developing applications, the DebugLog file is a useful place to begin tracing the activity that lead to a crash. By default this is \PENPOINT.LOG on the boot device.

DebugLogFlushCount 3.5.6

DebugLogFlushCount allows you to specify the number of characters that can be written to the debugger stream buffer before it is flushed. Normally this line is commented out. The default value is arbitrarily large, depending on current usage of the file system. However, if you are debugging a program that crashes after a write to the debugger stream, yet before the stream gets written to the log file, you can set this to a small value. When the value is set to 0, text is flushed as soon as it is written to the debugger stream.

DebugSet

3.5.7

DebugSet allows you to set one or more debugger flags to affect the behavior of applications. To specify a flag, follow the string "/D" with the flag identifier. For example, you specify the debug flag `B800` with the line:

```
debugset=/DB800
```

To specify more than one flag, separate the flags with one or more spaces. You must specify all flags within one DebugSet line; if your ENVIRON.INI file contains more than one DebugSet line, PenPoint uses the first one in the file.

See \PENPOINT\SDK\INC\DEBUG.H for a list of what subsystem uses what flag. The header files for many subsystems list the effects of setting the various bits in the flags.

You can examine and set the debug flag settings from the Debug Window accessory. You can also press [Pause] to go to the mini-debugger, and type **fl** to list the current flag values and **fs** to set a flag. For example, **fs F 30** sets the F flag to 30.

Other interesting flags are:

***1** When used with the _PP.OS (debug version of the PenPoint kernel), directs the heap manager to validate the heap after allocate and free calls (at a performance cost of about 15 percent). This flag is on by default in ENVIRON.INI.

D10000 Disables the mini-debugger (production and debug versions of PenPoint).

D40000 Disables the use of [Ctrl][C] and [Pause] keys to exit to the debugger.

G2 Allows page faults during scavenging to enter the debugger.

PenPointPath

3.5.8

Specifies the path to the PENPOINT directory within the boot volume (**theBootVolume**). This is useful for maintaining two separate PenPoint systems on the same hard disk. For example, you might maintain one version of PenPoint in \SYS1\PENPOINT... and another version of PenPoint in \SYS2\PENPOINT.... To use the first system, you would specify `PenPointPath=\sys1`; to use the second system, you would specify `PenPointPath=\sys2`.

PenProxTimeout

3.5.9

PenProxTimeout specifies the number of milliseconds between the pen going out of proximity and when the out of proximity event is actually sent to the input system. This delay helps to distinguish between actual out of proximity events and the user's hand wobbling. The default value is 350 milliseconds.

You can shorten (or lengthen) PenProxTimeout by changing "Gesture Timeout" from the Settings Notebook. (PenProxTimeout = `max(100, GestureTimeout/2)`. The default setting of gesture timeout, 600 milliseconds, results in PenProxTimeout being 300 milliseconds (same as it used to be). Many people can comfortably use the system with gesture timeout set to .4 seconds, this results in gestures being handled 1/10th of a second faster.

ScreenHeight 3.5.10

Specifies the height of the screen in device units.

ScreenWidth 3.5.11

Specifies the width of the screen in device units.

StartApp 3.5.12

StartApp specifies the first PenPoint application that PenPoint runs when it starts up. Usually this application is the Bookshelf. If you are writing a turnkey application, you might want to name your own application in StartApp, which would make it the only application running on the machine.

StealMem 3.5.13

StealMem tells PenPoint to use less memory, so that you can test how your application or service behaves in low-memory conditions. The value specified with StealMem is in hexadecimal. The line `StealMem=100000` causes PenPoint to "lose" 1MB of memory.

SwapBoot 3.5.14

SwapBoot allows you to tell PenPoint to boot by reloading its memory from information in the swap file, rather than reloading code. This is called a "swap boot."

To enable swap booting, add `SwapBoot=2` to ENVIRON.INI.

SwapFileSize 3.5.15

SwapFileSize specifies the size of the swap file in hexadecimal bytes. The suggested size is 5 MB (or 500000 hex). Be careful when reading this number; 500000 hex equals 5,000,000 decimal. The swap file is created in \\BOOT\PENPOINT.SWP. If SwapFileSize is set to 0, or is not specified, PenPoint does not create a swap file.

PenPoint swaps pages of data to the swap file. Code is not swapped, it is loaded from the loader data base or boot volume as needed.

TZ 3.5.16

TZ specifies the time zone with a string formatted according to the TZ environment variable used in the ANSI time routines. For complete information about the syntax of the TZ line, see Section 1.4, The "TZ Environment Variable," in the *Watcom C Library Reference for PenPoint.*

Version 3.5.17

Specifies the PenPoint version label. This label is created by GO and contains copyright and version information. Do not change this line.

VolSel 3.5.18

VolSel specifies the label of the volume that will contain the PenPoint directory hierarchy. You can see the label of a DOS volume when you use the DIR or CHKDSK commands.

From applications you can find the volume specified in VolSel with the well-known UID **theSelectedVolume.**

If your ENVIRON.INI file does not specify a `VolSel` environment variable, PenPoint uses **theBootVolume**.

If you specified a configuration with the Config option, do not modify VolSel.

WinMode 3.5.19

WinMode specifies the initial orientation of the screen device. There are four possible orientations, which correspond to the four possible rotations of a rectangular screen. The possible values are `PORTRAIT`, `LANDSCAPE`, `PORTRAIT_REVERSE`, and `LANDSCAPE_REVERSE`. In `PORTRAIT` orientation, the long axis of the screen is vertical; in `LANDSCAPE` orientation, the long axis of the screen is horizontal. In the `_REVERSE` orientations, the screen image is rotated 180 degrees from the normal orientation.

You usually use `LANDSCAPE` when running PenPoint on a PC.

ZoomMargin 3.5.20

ZoomMargin specifies the distance from the bottom of the screen to the bottom of the document opened with AutoZoom.

If you set ZoomMargin to 0 and specify ZoomResize as `FALSE`, the document that is opened with AutoZoom is locked into the page. If the document is a notebook, the user will not be able to close or resize the notebook. If the document is a page-based application, the user will not be able to turn away from the application.

ZoomResize 3.5.21

ZoomResize specifies whether the document opened with `AutoZoom` has a resize tab. Possible values are `TRUE` and `FALSE`. `TRUE` means that the document has a resize tab; `FALSE` means that the document does not have a resize tab.

If you set `ZoomMargin` to 0 and specify `ZoomResize` as `FALSE`, the document that is opened with `AutoZoom` is locked into the page.

Running in Tablet-Like Mode 3.6

When you set `Config=Tablet`, PenPoint runs in a configuration similar to that of the initial NCR and IBM pen computers. However, there are some considerations to using this configuration:

- **theSelectedVolume** is the drive on your PC from which you booted PenPoint. The bookshelf's contents live in \PENPOINT\SYS (its DOS name is \PENPOINT\SS) on the hard disk. Installed services, fonts, configured service instances, and so on are also stored on hard the disk.
- PenPoint won't let you browse **theSelectedVolume** in the directory view of the Connections notebook. In production PenPoint this prevents users from inadvertently moving, renaming, or deleting files that should be known only to PenPoint. If you want to see the hard disk in the directory view, set debug flag `B to 800`.
- Similarly, the Installer in the Settings notebook cannot see the hard disk; however, you can still install applications using the Connections notebook.

Before you run in `Config=DebugTablet` or `Tablet` mode, you need to:

- Remove the \PENPOINT\SS directory hierarchy from your hard disk. This gets rid of PenPoint state saved previously.
- Delete \PENPOINT\PENPOINT.IDX.
- Delete \PENPOINT\PENPOINT.DIR.

The first time you run in this configuration, you won't have any of these files.

Now boot PenPoint. The first time you boot, PenPoint will cold boot as usual. However, the loader will build its loader directory under \PENPOINT\SS\LR, which slows booting considerably. PenPoint marks the file system so that the next time you run PenPoint, a warm boot will occur. All subsequent times, booting will be faster.

Note that under this configuration, all installed EXEs and DLLs come from \PENPOINT\SS\LR. If you modify a DLL or EXE and want to replace it, you will either have to start with a clean disk, use the Upgrade utility, or copy your file into the \PENPOINT\SS\LR directory.

If you run in a tablet-like configuration and then decide to perform a cold boot or go back to the RAM configuration, you must clean up your disk. To do this:

- Delete \PENPOINT\PENPOINT.DIR.
- Delete \PENPOINT\PENPOINT.IDX.
- Remove the directory tree under \PENPOINT\SS.
- Remove the directory tree under \PENPOINT\SI.
- Remove the directory tree under \PENPOINT\OI.

On an actual tablet, when PenPoint loads code (services and applications) into the loader database, it *deletes* the \PENPOINT\APP directory and all application directories underneath it.

Even if you set `Config=Tablet`, the PenPoint running the SDK does not do this. If it did delete \PENPOINT\APP, it would mean you would have to reinstall the SDK to get \PENPOINT\APP back!

On a real tablet computer, the B2 debug flag controls this. *Do not set B2,* unless you want to reload the SDK. You can set it in ENVIRON.INI if you really, really want to get a configuration that closely matches a pen computer.

BOOT.DLC 3.7

The file \PENPOINT\BOOT\BOOT.DLC lists all the system dynamic link libraries that PenPoint needs to operate. A Dynamic Link Library is a body of code that programs can access at run time without needing to be statically linked into each program. It promotes code sharing and reduced executable size. PenPoint uses the same DLL file format and similar technology as OS/2 and Microsoft Windows.

PenPoint takes each name in BOOT.DLC, looks for a .DLL file of the same name in \\BOOT\PENPOINT\BOOT\DLL, loads that .DLL and tries to call an initialization routine in the DLL called **DllMain**. These initialization routines create classes, create objects and perform other initializations. For example, the window system initializes the screen and the input system starts up several subtasks and interrupt subtasks.

There are a standard set of DLLs which implement subsystems of PenPoint. The window system, UI Toolkit, handwriting translation, Application Framework, search and replace, and so on, are all implemented as DLLs.

PenPoint loads and initializes each DLL in order, so the ordering of BOOT.DLC is significant.

The main reason for modifying BOOT.DLC is to enable DB, the PenPoint source-level debugger. The first two lines in BOOT.DLC load the DLLs for DB:

```
#go0-cdb0-v2(0)          \\boot\penpoint\boot\dll\cdb0.dll
#go-cdb3-v2(0)           \\boot\penpoint\boot\dll\cdb3.dll
```

If you want to use DB, uncomment both these lines. CDB0.DLL is the Ring 0 portion of the debugger; CDB3.DLL is non-privileged code.

If you are only testing a restricted configuration, you can remove some DLLs; however, this can lead to unpredictable results.

If you have a DLL that you want loaded *independent* of your application, you can mention it in \PENPOINT\BOOT\BOOT.DLC. This should be rare: only system-wide DLLs such as input, windows, and so on, need to be listed in here. The order of DLL loading can be significant, so it's best to put your DLL at the end of the list.

You should not need to put your application's DLL file in BOOT.DLC, because your DLL is only needed if the user installs one or more applications which require it. Instead, you put a .DLC file in your application's directory which tells PenPoint which DLLs the application needs.

If you have a DLL that you want loaded *independent* of your application, you can mention it in \PENPOINT\BOOT\BOOT.DLC. This should be rare: only system-wide DLLs such as input, windows, and so on, need to be listed in here. The order of DLL loading can be significant, so it's best to put your DLL at the end of the list.

CONSOLE.DLC 3.8

The file \PENPOINT\BOOT\CONSOLE.DLC allows you to see debugging information while booting PenPoint on a single-screen system. Once PenPoint is booted, you can use the System Log application to view debugging information. You will want to use CONSOLE.DLC when debugging the behavior of your application during a boot—before you can run the System Log application.

To use CONSOLE.DLC, modify your MIL.INI file so that it specifies `MonoDebug=off`.

If the line is commented out, doesn't exist, or specifies anything other than "off," debugging information is not sent to your single screen.

SYSCOPY.INI 3.9

Once PenPoint knows the location of **theBootVolume** and **theSelectedVolume**, it can copy files from other disks to the directory that contains files for the running system. The file SYSCOPY.INI tells PenPoint which files to copy and where to place them.

The files listed in SYSCOPY.INI are those whose locations should not be hard-coded into PenPoint. Because the locations of these files are specified in SYSCOPY.INI, all you have to do is modify SYSCOPY.INI to use a different file. SYSCOPY.INI specifies the locations of the files listed in Table 3-6.

Table 3-6
SYSCOPY.INI Files

File	Used For
SYSAPP.INI	List of system applications to load at boot time (non-deinstallable).
APP.INI	List of general applications to load at boot time (deinstallable).
SERVICE.INI	List of services to load at boot time.
PENPOINT.RES	The PenPoint resources file.
DICT	The dictionary files to load at boot time.
FONT	The font files to load at boot time.
PREFS	The preferences to load at boot time.
HWXPROT	The handwriting prototypes to load at boot time.
PDICT	The personal dictionary files to load at boot time.

SYSAPP.INI 3.10

The file \PENPOINT\BOOT\SYSAPP.INI lists the system applications that must be present for PenPoint to run correctly. This list includes:

```
\penpoint\boot\app\Section
\penpoint\boot\app\Table of Contents
\penpoint\boot\app\Notebook
\penpoint\boot\app\MiniText
\penpoint\boot\app\Settings
\penpoint\boot\app\Helpnb
\penpoint\boot\app\inboxnb
\penpoint\boot\app\oboxnb
\penpoint\boot\app\statnb
\penpoint\boot\app\Connections
\penpoint\boot\app\Keyboard
\penpoint\boot\app\Placeholder
\penpoint\boot\app\Accessry
```

Applications loaded with SYSAPP.INI cannot be deinstalled. If you are providing a turnkey system, you can load your specific applications from SYSAPP.INI.

APP.INI 3.11

The file \PENPOINT\BOOT\APP.INI specifies the installable applications to load at boot time. These applications can be deinstalled at a later time.

When testing an application, it is usually easiest to add the application to APP.INI. That way the application is installed each time the system is booted.

Setting Up Specific Configurations 3.12

The preceeding sections described the initialization files individually. However, when you change your configuration, you will need to make changes to several files.

This section describes the modifications that you make to support various devices.

One or Two Monitors 3.12.1

When you run PenPoint on a PC, you must have a VGA display on your machine. PenPoint can display the entire Notebook user interface on the VGA monitor.

If you have two monitors (one VGA and one monochrome), PenPoint detects that there are two monitors (at boot time) and will simultaneously display the PenPoint Notebook User interface on the VGA screen while displaying debugging output on the monochrome display.

If you have a single monitor and need to view debugging output while booting PenPoint, add a line to your MIL.INI that specifies `MonoDebug=off`. Such a line exists in the _MIL.INI shipped with the PenPoint SDK, but it is commented out.

Configuring a Mouse 3.12.2

The SDK version of PenPoint is is set up for a Wacom pen tablet. If you have a mouse, you must modify your MIL.INI file to specify a mouse. PenPoint supports four types of mice: Microsoft compatible mouse, Logitech C7/C9 mouse, bus mouse, PS/2 mouse.

To attach a Microsoft mouse, add a `UNIPENTYPE=Microsoft` line to your MIL.INI, specifying the serial port to which the mouse is connected. To attach a Logitech mouse, add a `UNIPENTYPE=Logitech` line to your MIL.INI, again specifying the serial port to which the mouse is connected.

To use a PS/2 mouse, add the line `PS2Mouse=on` to your MIL.INI.

Finally, if you are left handed, you might want to add the line `LeftyMouse=true` to your MIL.INI.

Configuring a Digitizing Tablet 3.12.3

To use a digitizing tablet with PenPoint, you must:

- Add a line to your MIL.INI that specifies the tablet and the serial port to which the tablet is attached.
- Configure the serial port with a DOS MODE command before running PenPoint. If the digitizing tablet is attached to COM1, the mode command is:

  MODE COM1 96,0,7,2

Booting PenPoint on a PC 3.13

To start PenPoint, run GO.BAT by typing **`\penpoint\sdk\util\dos\go`**. GO.BAT is a batch file which sets up your machine and then runs PPBOOT.EXE.

You may need to modify GO.BAT to:

- Configure a serial port for the digitizing tablet.
- Load a Logitech or Microsoft mouse driver.
- Configure a serial port for low-level MIL debug output.
- Unload memory managers and TSRs.
- Run some disk mirroring or FAT preserver utility.

GO.BAT contains a number of lines that are commented out to remind you of these possible additions.

If you are testing a new application and the application goes badly wrong, PenPoint may damage the root directory of your boot volume. This is more likely to be a problem if you run with `Config=DebugTablet` or `Tablet` (so that the selected volume set to your hard disk). You may want to use the Norton Utilities' format recover (**`fr`**) with the save (**`/save`**) option or some other format save utility to save your hard disk's format information before starting PenPoint. If you have this utility, add **`fr /save`** before `ppboot` in GO.BAT and `chkdsk` after. If CHKDSK reports an error, run `fr` to fix it.

PC Tools has a similar MIRROR capability.

The hierarchy of applications on the bookshelf files its state in \PENPOINT\SYS\BS on the selected volume. If you are running PenPoint on a PenPoint computer, or if you set `VolSel` to a disk on a PC, the previous state is still around. The previous configuration of the Notebook should reappear. This is what happens when you power off and power on a PenPoint computer.

If you are running PenPoint on a PC from memory, there's nothing in memory when you boot PenPoint. Therefore \PENPOINT\SYS will be blank. However, during boot, the default \PENPOINT\BOOT\SYSCOPY.INI instructs PenPoint to copy \PENPOINT\SYS from the boot volume to the selected volume. Thus you can set things up so that a previous saved configuration of the Notebook reappears when you boot from disk. This is how the default Notebook appears when you boot a PenPoint computer—Notebook state was filed and then copied to the boot diskettes.

Loading Debug PENPOINT.OS 3.13.1

GO.BAT runs \PENPOINT\BOOT\PPBOOT, which in turn loads MIL.OS, MIL.INI, and PENPOINT.OS. If you want to load core PenPoint with `DEBUG` information in it (to get, for example, symbolic names for Class Manager objects), you need to comment out the line

```
\penpoint\boot\ppboot \penpoint\boot
```

and uncomment the line

```
\penpoint\boot\ppboot \penpoint\boot\_pp.os \penpoint\boot\mil.os \penpoint\boot\mil.ini
```

What Happens During Booting 3.13.2

PPBOOT prints the names of the files that it loads.

If you've enabled `LowLevelDebug` in MIL.INI, you'll see debugging information on your second monitor or serial port.

The MIL prints out some memory statistics and then attempts to load PenPoint.

The boot program displays a PenPoint logo and pen. It displays numbers in the lower corners of the screen:

- In the lower left is the version of the PPBOOT boot program.
- In the lower right is a number indicating whether PenPoint found a swap file:
  - **0** No swap file found.
  - **1** Swap file found but the boot program couldn't use it (delete \PENPOINT.SWP and try again).
  - **2** Old-format swap file found (historical, you should never see this).
  - **3** Found a swap file, using it.

As PenPoint boots, the pen fills with black. After a while, PPBOOT transfers control to PenPoint, which proceeds to read ENVIRON.INI, BOOT.DLC, SYSAPP.INI, and so on.

After a while (two to four minutes on a 386 PC) you should see the small clock appear, followed by the standard bookshelf and Notebook configuration.

If PenPoint doesn't work, press [Pause] then enter **q**. Look in \PENPOINT.LOG on your boot volume for errors.

Boot Error Messages 3.13.3

Here are some of the errors that you will probably see during booting, together with explanations:

The first time PenPoint runs, it will complain about a missing PENPOINT.SWP file. This is normal—the swap file does not exist until the second time you run PenPoint.

If you load _PP.OS, you get some additional warning messages about `No init routine provided for` *file*`.dll`.

You might see several `Int w/o RB: 7` messages during booting. These are related to parallel port interrupts, are relatively benign, and are explained later in this chapter.

To save memory, PenPoint compacts processes (such as the **processCount** 0 appMonitor processes). Messages about this are not in error.

Broken Pen During Booting 3.13.4

If there's a problem during booting, a broken pen icon appears on the PenPoint boot screen. The most common value is 1000, which means "you haven't enabled any pointing device in your MIL.INI."

The error codes are in the file \PENPOINT\SDK\INC\BOOTERRS.H as shown here:

```
Error codes used with the AbortBoot routine.
   These codes are displayed when a fatal error occures during booting.

   Code zero is never used.
   Codes 1 through 999 are reserved for the boot program. Of these 1 through
   99 are defined by GO and consistent across all machines. Codes 100 to
   999 can be used by OEMs for machine specific errors.
   The portable codes used by PenPoint start at 1000.
   Naming convention: to make it easier to track-down the location of
   the error code a name is constructed thusly:
   "bootErr" + <file name> + <terse description>
   See below for examples.
```

```
 Errors in the boot program
#define bootErrDriveNotFound            1
#define bootErrOpenDrive                2
#define bootErrLabelNotFound            3
#define bootErrCorruptFileSystem        4
#define bootErrMILNotFound              5
#define bootErrMILReadErr               6
#define bootErrMILWrongFormat           7
#define bootErrMILWrongSignature        8
#define bootErrMILWrongTailLen          9
#define bootErrMILZeroLength            10
#define bootErrMILTooBigForMem          11
#define bootErrMILIniNotFound           12
#define bootErrReadErrMILIni            13
#define bootErrPPNotFound               14
#define bootErrPPMILVersionMismatch     15
#define bootErrPPReadErr                16
#define bootErrPPWrongFormat            17
#define bootErrPPWrongSignature         18
#define bootErrPPWrongTailLen           19
#define bootErrPPZeroLength             20
#define bootErrPPTooBigForMem           21
#define bootErrSwapFileReadErr          22
#define bootErrMemSizing                23
#define bootErrNoLinearPgDirEntry       24
#define bootErrInitFileSys              25
#define bootErrUnknownDrive             26
#define bootErrExitProgram              27


 Errors in PenPoint

// Used in DrvMILSetWellKnownIds()
#define bootErrDrvmilNoStylus           1000
#define bootErrDrvmilNoPowerDev         1001
#define bootErrDrvmilNoTimer            1002
#define bootErrDrvmilNoClock            1003
#define bootErrDrvmilNoIntrCtlr         1004
#define bootErrDrvmilNoScreen           1005
#define bootErrDrvmilNoBlock            1006

// File system related
#define bootErrStartupNoBootDiskLabel   1007

// Non-maskable interrupt
#define bootErrDrvmilNMI                1008

// StdError: Note also displays
// TagAdmin(status) and TagNum(status),
#define bootErrStdErrorOccurred         1009

// Environ.ini file not found
#define bootErrStartupNoEnvironIni      1010

// Boot volume not found (or error accessing)
#define bootErrFsvolNoBootVolume        1011

// Invalid PenPointPath specified in environ.ini
#define bootErrFscInvalidPenPointPath   1012
```

```
// Boot.dlc file not found
#define bootErrConfNoBootDlc            1013
// Error creating swap file
#define bootErrVMSwapFile               1014

// Error starting page daemon
#define bootErrVMPageDaemon             1015

// Not enough memory to boot
#define bootErrVMMemory                 1016

// Boot program did not pass in valid boot time
#define bootErrDrvmilNoBootTime         1017

// Boot program did not pass in complete V8086 information
#define bootErrDrvmilNoV8086Info        1018

// A core MIL device (timer, interrupt, debug) failed to initialized
#define bootErrDrvmilCoreInitFail       1019
```

If code calls **StdError**, **StdSystemError**, **StdUnknownError** or **StdMsg** before booting is complete, the broken pen displays with the error number of the standard error. The root window isn't displayed until booting is complete, so the notes that these normally put up are not visible.

Broken Pen Error 1008 3.13.4.1

Error 1008 indicates that a non-maskable interrupt (NMI) occured and was not handled by a MIL NMI device. On a PC NMI is used for two purposes: memory parity error and (in some cases) emulation on video cards.

A parity error indicates the failure of a memory chip (soft or hard). You should run a memory diagnostic when the machine is good and hot—partially (30%) blocking the fan is a way to do this. The other possibility is that you are using an older ATI video card. They emulate other types of video cards by generating an NMI when the (nonexistent) registers of the emulated card were accessed. The NMI service routine then emulates the hardware in the video BIOS. The ATI card has a switch which "disabled advanced video modes" and prevented these NMIs. Check to see if your video card uses NMI or supports some form of "emulation."

When the NMI occurs is a big clue as to what was tickled to cause it (for instance, what DLL was loading, and so on).

Broken Pen Errors between 100 and 999 3.13.4.2

You may see other boot error numbers. such as 115 or 126, that are not defined in BOOTERRS.H. The boot program overloads the error number by tacking on the number of the drive it is booting from. 0 for floppy and 100 for hard disk. Thus 115 means error 15 when booting from the hard drive.

Using PenPoint on a PC 3.14

For more information on working in PenPoint, read the end-user documentation.

Using a Mouse 3.14.1

The Pen can detect not only stylus tip up and down, but also moving the pen in and out of **proximity**. Handwriting and gesture translation software uses this to figure out when to convert ink dribbles. The mouse does not have any notion of in and out of proximity, so to simulate going out of proximity, you click the middle button. In general, after handwriting or making any gesture with the mouse, "lift the pen away" by clicking middle.

Parallel Port Interrupts 3.14.2

Because of a problem in the 8259 programmable interrupt controller (PIC) used by most PCs, some machines can generate sporadic interrupt 7s under certain, unpredictable conditions. The symptom is that you see several `Int w/o RB: 7` messages at boot time. We have not seen this behavior on tablet hardware.

PenPoint attempts to avoid hanging conditions by disabling interrupt 7 whenever too many bad interrupts occur, and the re-enabling interrupt 7 at a later time. This does not limit PenPoint's functionality.

Theoretically, parallel port I/O could become impossible, if PenPoint were to constantly disable and enable interrupt 7. However, we have not seen this situation.

Installing an Application 3.15

While most operating systems require the application developer to provide installation software (or, at a minimum, copy the application and its files to disk), PenPoint provides an Installer, which makes installation of all installable software consistent.

The installer expects application files to be in specific directories on the distribution diskette. The installer copies files from those directories to directories in the RAM file system.

There are two ways you can install an application:

- Use the Installer to install your application while PenPoint is running.
- Add the file name to the APP.INI file so that PenPoint installs your application at boot time.

You can also mark a volume so that the Installer is invoked automatically when the disk is mounted. You do this by turning to the disk options card in the connections notebook. When you set Quick Installer to Yes, the Initial View field becomes active. From in this field, you can select whether the installer should be activated for applications, services, and so on when the disk is mounted.

The Installer can do lots of other things for your application, such as copy stationery, help files, bitmaps for icons, and such to the PenPoint computer. The Tic-Tac-Toe application, described in this manual, uses some of these features. These features are also documented fully in *Part 12: Installation* of the *PenPoint Architectural Reference.*

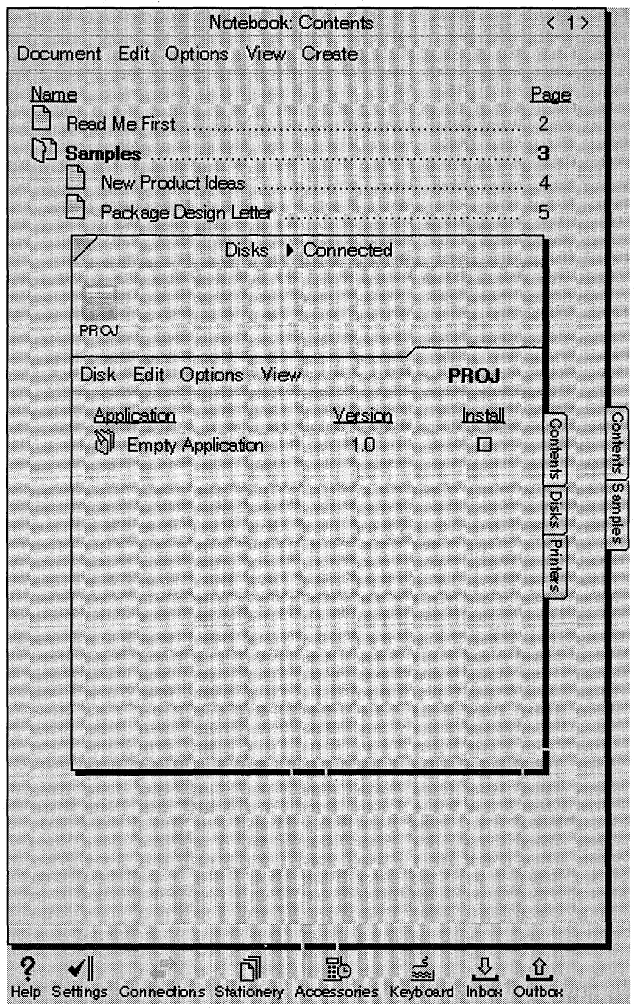
Installing an Application While PenPoint is Running 3.15.1

PenPoint is designed to not need rebooting. Users can go forever without having to reload PenPoint and applications from disk. Thus, there needs to be some way for the user to install new applications while PenPoint is running. There is: it involves using the Installer application.

The Installer figures out what applications can be installed by searching for subdirectories of \PENPOINT\APP on all known volumes. So, if you copy your application's executable and its supporting files to a subdirectory of \PENPOINT\APP on some volume, you will be able to install it while PenPoint is running, just as a user who has purchased your application will install it. In the case of EMPTYAPP, create a directory on your hard disk or on a floppy (with a volume label) called \PENPOINT\APP\EMPTYAPP, and copy EMPTYAPP.EXE to it. (The make file for empty app does this for you.)

Open the Connections notebook, turn to the disks page, and tap on your disk. Choose Applications from the View menu. Empty Application should appear in the list of applications on the disk. Tap on the installed box next to Empty Application.

Figure 3-1
Installing an Application



In this figure, the application installer has found an application directory called Empty Application in \PENPOINT\APP on the disk labelled PROJ.

The application name is the name of its directory in \PENPOINT\APP. PenPoint supports longer file names than DOS. A rule in the Makefile gave the EMPTYAPP directory and .EXE longer PenPoint names using the STAMP utility.

Boot-Time Install

3.15.2

You can get your application installed as part of the boot process by adding it to \PENPOINT\BOOT\APP.INI.

APP.INI controls which applications are installed during boot. Each line in APP.INI specifies the PenPoint file name of an application directory. You must specify the volume and the entire path to the application directory. However, lines in APP.INI do not specify the executable file itself. PenPoint copies several different files and

To speed up the boot process while testing, you can remove applications you don't wish to use from APP.INI (if there are any).

directories from the application directory to the running application directory (such as help, stationery, and so on).

Thus, for your own project **Project Scheduler** you would add the following to \PENPOINT\BOOT\APP.INI:

```
\\boot\penpoint\app\Project Scheduler
```

"System" applications are mentioned in a separate file called \PENPOINT\BOOT\SYSAPP.INI. You should be careful when modifying this file—some of these applications are required for the Notebook environment to work.

Whether an application appears in the Accessories notebook or in the Stationery notebook is determined by a flag in the arguments you send to the Application Manager when you initialize your application class (when the user installs your application). For more information, see the description of **clsAppMgr** in *Part 2: Application Framework* of *PenPoint Architectural Reference, Volume I.*

Application .DLL and .DLC Files 3.15.3

If you create multiple applications that use the same classes, you can save memory by placing the shared classes in a common DLL (dynamic link library).

When PenPoint is instructed to load an application (either in response to the Installer or when reading APP.INI), it first finds the directory whose name matches that of the application.

PenPoint then looks in the directory for a file with the same name as the directory, but with a .DLC extension. A .DLC file contains a list of executable and DLL files that the application requires. If your application requires a class that is in a common DLL file, it should list the DLL file in its .DLC file.

If PenPoint finds the .DLC file, reads the names of the DLLs in the file and searches for them in the loader database. If a DLL is not found in the loader database, PenPoint loads it. PenPoint also calls **DllMain** in the new DLL (if it exists), thus providing each DLL with a standard initialization technique. **DllMain** can create classes, create objects, and perform other initializations. The last line in the .DLC file must list an executable with the same name as the application directory (with the extension .EXE).

If there is no .DLC file, PenPoint then searches the application directory for an executable with the same name as the application directory (with the extension .EXE).

The name of the .DLC must match the PenPoint name of the directory. If you give the application directory a long PenPoint name, but forget to change the .DLC or .EXE name (with the STAMP utility), installation will fail.

Executing the Application 3.16

The easiest way to start an application is to create a **document** for that application from the Notebook and then turn to or float that document. You can create new documents in several different ways:

- Open Accessories, and tap on the icon of an accessory application.
- Choose an application from the Create menu in the Notebook's table of contents. This creates a new document on the Notebook.
- Make a caret ∧ gesture over the Notebook and choose the application from the stationery menu. This creates a new document at the location where you made the gesture.
- Make a caret ∧ gesture over the bookshelf and choose the application from the stationery menu. This creates a new document in the bookshelf.
- Open the Stationery notebook and use the copy gesture to copy a piece of stationery to the main Notebook or elsewhere. Some applications provide several different kinds of **stationery**, so the user can pick a particular kind of document to start from.

Different applications allow different kinds of document creation. (Appearance in the Stationery notebook is controlled by the **appMgr.flags.stationery** flag, and appearance in the Accessory notebook is controlled by the **appMgr.flags.accessory** flag; you set these in your application's init routine). It's convenient during testing to allow both kinds of creation.

Volume Selection 3.16.1

On a PC you have a choice. If you select your hard disk, the state of your simulated PenPoint notebook (documents, table of contents, and so on) will be retained after you quit PenPoint or power off, just as on a PenPoint computer. On the other hand, there is a slight risk that during development your applications won't file correctly and hence will cause trouble when you restart PenPoint. Running PenPoint from RAM avoids these problems, but presents you with an empty Notebook every time you restart it.

Interrupting PenPoint 3.16.2

On a PC, the Pause key interrupts the current task. It puts you either in a mini-debugger built into PenPoint, or into DB if you are running the PenPoint Source Debugger. In either case, type **g** to continue or **q** to quit and return to DOS. (Of course, you can only return to DOS on a PC).

Exiting PenPoint 3.16.3

There are two ways to exit from PenPoint:

- Tap on Settings, navigate to Preferences-Power, and tap the Shutdown button. Confirm that you want to power-off PenPoint.
- Press [Pause] and enter **q**.

Note that either way, PenPoint takes time to shut down because it has to flush all cached information to disk before halting.

Depending on your machine configuration, your PC may return to DOS or may hang. GO has found that network software, spoolers, TSR's, disk caches, and RAM drives can all affect this. If your machine does not return to the DOS prompt, reboot.

When you get a DOS prompt, you should run CHKDSK to check for file system errors.

More on the Bookshelf 3.17

If you're running PenPoint on a PC and your root volume is your hard disk, there will be lots of files in \PENPOINT\SYS\BS. This directory contains all notebooks and all their files. You should be able to find your documents' directories in BS. Within each document directory, you'll find a DOC.RES file, which contains the document's objects and data.

You can inspect this directory in PenPoint by setting the B debug flag to hexadecimal value 800 in ENVIRON.INI.

After exiting PenPoint, you'll find that PenPoint has left PENPOINT.DIR files in \PENPOINT and in its subdirectories. The PENPOINT.DIR files map PenPoint file system names to the DOS files and contains attribute information about the files.

Using the Notebook 3.17.1

The Notebook initially displays its table of contents view. This is always at the front of the Notebook.

You can page through applications by tapping the arrows to the left or right of the page number. You can also flick backwards and forwards through the pages by drawing a short line to the left or right in the title bar.

You can also jump to a page by tapping the button in its page number in the table of contents. If you double-tap on the document's button, it is "torn out" from the Notebook and floats on top of the Notebook (if you enable this in the Float & Zoom portion of Settings).

For more information on using PenPoint, read the end-user manual, *Getting Started with PenPoint.*

The Universal Serial Pen Driver 3.18

After writing a number of PenPoint digitizer drivers for devices that connect through a serial port, it became apparent that they were all pretty much alike. These pointing devices want to pass three or four pieces of information into the computer: x position, y position, tip-down and (optionally) pen-in-proximity-to-screen.

Because this amount of information is difficult to pack into a single byte most serial pointing devices opt for a multi-byte format. Some bit or pattern is reserved to signal the start or end of the multi-byte frame, the tip and proximity states each get a bit somewhere in the frame, and the bits which make up the X and Y position are given as many bits (in as many bytes) as they need. Some devices (like mice) are relative and pass small (less than 8-bit) deltas. In contrast, digitizers return absolute coordinates, which require 10-13 bits of space in binary formats or 4-5 bytes or characters in ASCII formats.

Unfortunately, each manufacturer that needed such a protocol invented their own. Later products (such as the Wacom) can be programmed to emulate many of the pre-existing products and protocols.

The universal pen driver is an attempt to eliminate the need for a custom driver for every digitizer and mouse in the world. This driver parses a simple language that describes the protocol of a serial pointing device. The driver also contains a number of "canned" or predefined descriptions of some popular products.

The key expressions in this language usually contain:

- The relative position of a byte in the frame.
- A mask to AND the byte with.
- A value to compare the result with.

Thus if bit 3 in the second byte is zero when the tip is touching we describe this as `T,1,4,0`; 1 for the byte (bytes are numbered from 0), 4 for masking bit 3, and 0 as the expected result when the tip is down. The expression `P,0,9,8` says that the pen is in proximity to the screen when byte 0 contains: 0 in bit 0 and 1 in bit 3 (1001B mask compared with 1000B).

Numeric values are described in a similar way:

- A byte number
- A mask
- An offset
- A scaling factor

To support values whose bits are split across multiple bytes, UniPen sums the frame expressions that have the same tag. For example, in the Bit Pad 2 binary format, the 12 bits of the X position are contained in the low-order 6 bits (0 through 5) of the second and third bytes (bits 0 through 5 come from bits 0 through 5 of the second byte; bits 6 through 11 come from bits 0 through 5 of the third byte). We describe this with the string:

```
X,1,63,0,1 X,2,63,0,64
```

The value 63 (0111111B) masks the low-order 6 bits; the scaling factor of 1 is a no-op, the scaling factor of 64 shifts the value 7 bits to the left before summing.

To describe the Bit Pad 2 ASCII format we use the offset to subtract the value of an ASCII "0" to convert a 4-digit ASCII string starting in the first byte:

```
X,0,127,-48,1000 X,1,127,-48,100 X,2,127,-48,10 X,3,127,-48,1
```

Other digitizers are left as an exercise to the student.

UniPen Command Syntax 3.18.1

Table 3-7 lists the UniPen commands.

Table 3-7
UniPen Commands

Command	Meaning	Definition
V	VERBOSE	Toggles the verbose mode used for debugging. The compile time switch VERBOSE must be enabled in the source code for this to work.
C,baud,parity,bits,stop	COMPORT	Sets the com port settings for the operating mode: baud = the baud rate parity ('O'= Odd, 'E'= even, 'N' = none); bits = the # of data bits (6,7,8); stop = the number of stop bits (1,2).
I,byte1,byte2,...	INIT	Specifies a sequence of bytes to be sent to the device for initialization. The sequence can be up to 16 bytes long.
G,baud,par,bit,stop,b1,...	ALTERNATEINIT	Specifies an alternate sequence of bytes to be sent to the device with a different com port setting. This is used when the device's initial com port settings are different than the operating settings (cmd-C). baud = the baud rate parity ('O'= Odd, 'E'= even, 'N' = none); bit = the # of data bits (6,7,8); stop = the number of stop bits (1,2); b1,... = the byte sequence to send. If the baud rate is zero, the sequence will be sent to the device at 1200,2400,4800, 9600 baud (four times). The alternate init ('G') happens before the INIT command ('I').
N,name	NAME	Specifies a name for the protocol. The name can be any sequence of characters except space or comma. If used, this must be the first command if used.
L,xLimit,yLimit	LIMIT	Sets the maximum value for x and y. These limits must be set properly for the proper operation of the pen. In a machine where the digitizer is not combined with the screen, these limits are used to scale the pen's position to fit the screen. This allows the use of digitizers and screens of different dimensions. For combined screen and digitizers (such as the SuperScript), these limits are used for coordinate tranformation when different screen orientations are chosen. If you loose pen and ink alignment when you change the orientation preference, these values are probably incorrect.

continued

Table 3-7 (continued)

Command	Meaning	Definition
T,byte,mask,compare	TIP	Defines an equation for the computing the tip state where: TIP = ((protocol[byte] & mask) == compare) ; byte = the byte index into the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with compare = value the masked byte is compared with.
P,byte,mask,compare	PROXIMITY	Defines equation for the computing the proximity state where: TIP = ((protocol[byte] & mask) == compare); byte = the byte index into the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with; compare = value the masked byte is compared with.
X,byte,mask,add,mult	X	Defines an equation for computing X where: X = X+ ((packet[byte] & mask) + add) * mult; byte = the byte index into the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with; add = the value added to the masked value; mult = the value multiplier. There can be more than one X command.
Y,byte,mask,add,mult	Y	Defines an equation for computing Y where: Y = Y+ ((packet[byte] & mask) + add) * mult; byte = the byte index into the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with; add = the value added to the masked value; mult = the value multiplier. There can be more than one Y command.
S,byte,mask,compare	SYNC	Defines a equation for checking the validity of the packet. If (packet[byte] & mask) != compare the packet is considered out of sync. byte = the byte index into the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with; compare = value the masked byte is compared with.
B,byteCount	BYTECOUNT	Defines the number of bytes in a packet.
E	EXTEND	Tells UniPen to sign extend the X and Y values from 8 bits to 16 bits.
J,byte,mask,compare	NEGATE X	Used to process sign bits in the protocol for X. X will be negated if (packet[byte] & mask) == compare. byte = the byte index into the protocol stream from 0 to the max byte - 1; mask = the value the byte is masked with; compare = value the masked byte is compared with.
K,byte,mask,compare	NEGATE Y	Used to process sign bits in the protocol for Y. Y will be negated if (packet[byte] & mask) == compare. byte = the byte index in to the protocol stream from 0 to the max byte -1; mask = the value the byte is masked with; compare = value the masked byte is compared with.
F,[XY]	FLIP X or Y	Inverts the coordinates of either X or Y. For example, F,X would result in X = xLimit - X being executed. F,Y would do the same for Y.

continued

Table 3-7 (continued)

Command	Meaning	Definition
R,num,res,byte1,byte2,...	RESOLUTION	Tells UniPen the resolution of the device and defines a sequence of bytes to initialize the mode. There must be at least one resolution command and there can be as many as three. If there is more than one resolution command, the byte sequence will be sent to the device whenever that resolution mode is to be used. If there is only one resolution command, the byte sequence is optional. num = the resolution number (0,1,2); res = the resolution in counts/meter; byte1,.. = the byte sequence used to initialize the mode.
Q,num,res,byte1,byte2,...	SAMPLE RATE	Tells UniPen the sample rate of the device and defines a sequence of bytes to initialize the mode. There must be at least one sample rate command and there can be as many as two. If there is more than one sample rate command, the byte sequence will be sent to the device whenever that sample rate mode is to be used. If there is only one sample rate command, the byte sequence is optional. num = the sample rate number (0,1,2); res = the number of samples per second; byte1,.. = the byte sequence used to initialize the mode.
D,scale	DELTA	Tells UniPen that the counts from the device are relative positions and not absolute positions. scale = the scale factor for the delta counts. (1 = no scale, 2 = 2x scale, and so on.)
A	ABSOLUTE	Tells UniPen that the counts from the device are the absolute position of the pen.
O,x,y	OFFSET	Offsets the x and y positions or defines the difference between logical zero and physical zero. A positive number will bring the "ink" closer to the lower left corner; a negative one farther away.
H,threshold	THRESHOLD	Defines the reporting threshold. The pen must move at least threshold counts before a new position will be reported. For most devices this can be one.
U	UNIFIED	The display and digitizer are a combined unit. When they are not, PenPoint scales the digitizer position to fit the screen based on the values provided by the Limit command (see above). Note that for accurate matching of the pen and "ink" you must specify ScreenPixelsPerMeter in MIL.INI to match your screen. This may be easily calculated by measuring the width of the actual pixels along the long (landscape) axis in centimeters. ScreenPixelsPerMeter = 640 / length * 100
M	MUNGE PROXIMITY	Supresses proximity sensing for any pre-defined type. Note that there is no reason to use this command when defining your own protocol, just don't specify a "P" tag.

Notes on Using the UniPen Driver 3.18.2

Use the `UNIPENPORT` tag in MIL.INI to set the communications port. For example, the first line sets UniPen to COM1:, the second line sets UniPen to the communication port at 2F8, IRQ 5:

```
UNIPENPORT = 1
UNIPENPORT = 760,5
```

Use the `UNIPENPORT` tag in MIL.INI to select a predefined protocol. This line sets UniPen to the Wacom 510C digitizer:

```
UNIPENTYPE = WACOM510C
```

The current predefined types are:

MICROSOFT Microsoft serial, two-button mouse.

LOGITECH Logitech C7 or C9 serial, three button mice. For a Logitech MouseMan use the MICROSOFT tag.

WACOM510C The switches under the control unit must be set as shown here (X = on, O = off):

```
              DS1       DS2       DS3
Front   OXOXXOOX  XXOXOXXO  XXOOXXOO   Back
```

WACOM510 The older Wacom units, red power LED, attached power cord. The switches must be set as shown here (X = on, O = off):

```
Front   OOOXOOXO  XXOXOXXO  XOOOXOOO  OOOOOOOO   Back
```

SuperScriptII The SuperScript II LCD/digitizer combo. For the ink to be aligned you must specify `ScreenType=SuperScript` or `ScreenPixelsPerMeter=3690` in MIL.INI.

CalCompDBII CalComp's DrawingBoard II, should also work with their "Wiz" product. Use the default "Hi Resolution Binary" format (#23) in "run" mode, 9600 baud no parity, eight data bits, one stop bit, 1000 lpi resolution, 125 pps. Enable "Send when out of proximity," button 10=1 in bank B.

AceCat5by5 A very low cost (under $150) 5" by 5" digitizer from AceCAD. Supports proximity, has corded pen. Draws its power from the PC keyboard connector or an optional external AC adapter.

These are generic descriptions. You will probable have to "tune" these by placing the parameters into MIL.INI and adjusting for the specific charatersitics of your digitizer.

GAZELLE For products from Gazelle System (now owned by Logitech).

MM The common "MM" digitizer protocol.

BITPAD2 BitPad 2 binary protocol.

BITPAD2ASC BitPad 2 ACSII protocol.

To define a new protocol in MIL.INI use the `UNIPENPROTOCOL` tag. Also, since protocol definitions can get long, we created several tags for defining protocols. It does not matter which commands go with which tags, because all of the tags are sent to one central parser. The tags are `UNIPENCOMPORT`, `UNIPENINITIALIZE`, `UNIPENRESOLUTION`, `UNIPENSAMPLERATE`, `UNIPENTIP`, `UNIPENPROXIMITY`, `UNIPENXPROTOCOL`, `UNIPENPROTOCOL`, `UNIPENPROTOCOL`. For example, the Microsoft serial mouse protocol is:

```
UNIPENTIP           = T,0,32,32
UNIPENPROXIMITY     = P,0,16,0
UNIPENXPROTOCOL     = X,0,3,0,64 X,1,63,0,1
UNIPENYPROTOCOL     = Y,0,12,0,16 Y,2,63,0,1 F,Y
UNIPENPROTOCOL      = B,3 S,0,64,64, L,3600,2400 E D,5 H,1
UNIPENRESOLUTION    = R,0,7800
UNIPENSAMPLERATE    = Q,0,30
```

The C,L,T,B,R,Q,H, at least one X and Y, and an A or a D command are required.

Other Sample Definitions 3.18.2.1

SuperScriptII

```
I,82,48,61,48,13,10 C,9600,N,8,1 T,0,1,1 P,0,64,0
X,1,3,0,16384 X,2,127,0,128 X,3,127,0,1
Y,4,3,0,16384 Y,5,127,0,128 Y,6,127,0,1
S,0,128,128 B,7 L,6911,5183 A R,0,39370 Q,0,100 H,1 U O,0,157;
```

AceCat5by5

```
C,9600,O,8,1 R,0,19500 Q,0,100 L,2437,2437
I,0,64,98 P,0,64,0 T,0,1,1 X,1,127,0,1 X,2,127,0,128
Y,3,127,0,1 Y,4,127,0,128 S,0,128,128 A H,1 B,5
```

Gazelle:

```
N,GAZELLE C,9600,O,8,1 R,0,16185 Q,0,50 T,0,1,1 P,0,64,64
X,1,127,0,1 X,2,31,0,128 Y,3,127,0,1 Y,4,31,0,128
S,0,128,128 L,3200,2530 O,210,300 B,5 A H,10;
```

MM:

```
N,MM C,9600,O,8,1 R,0,19500 Q,0,100 L,5000,5000
I,0,64,98 P,0,64,0 T,0,1,1 X,1,127,0,1 X,2,127,0,128
Y,3,127,0,1 Y,4,127,0,128 S,0,128,128 A H,1 B,5;
```

BitPad2ASCII:

```
N,BITPAD2ASCII C,9600,E,8,1 R,0,7800 Q,0,100
T,10,01,01 P,10,01,01 L,5000,5000
X,0,127,-48,1000 X,1,127,-48,100 X,2,127,-48,10 X,3,127,-48,1
Y,5,127,-48,1000 Y,6,127,-48,100 Y,7,127,-48,10 Y,8,127,-48,1
S,11,127,13 B,13 A H,1;
```

BitPad2:

```
C,9600,E,8,1 R,0,7800 Q,0,100 T,0,4,4 P,0,1,0
L,5000,5000 X,1,63,0,1 X,2,63,0,64 Y,3,63,0,1 Y,4,63,0,64
S,0,64,64 B,5 H,1 A;
```

LOGITECH:

```
C,9600,O,8,1 R,0,7800 Q,0,50 L,3600,2400
P,0,1,0  T,0,4,4  X,1,127,0,1 J,0,16,16 F,X Y,2,127,0,1 K,0,8,8 F,Y
S,0,128,128 D,5 H,1 B,3;
```

WACOM510C:

```
C,9600,O,7,2 R,0,9906 Q,0,100 T,0,4,4 P,0,1,0
L,2320,1510 X,1,63,0,1 X,2,63,0,64 Y,3,63,0,1 Y,4,63,0,64
S,0,64,64 B,5 H,1 A;
```

WACOM510:

```
C,9600,O,8,1 R,0,9906 Q,0,100 T,0,4,4 P,0,1,0
L,2320,1510 X,1,63,0,1 X,2,63,0,64 Y,3,63,0,1 Y,4,63,0,64
S,0,64,64 B,5 H,1 A;
```

CalCompDBII:

```
C,9600,N,8,1 R,0,39370 Q,0,125 T,0,4,4 P,3,32,0
L,7000,5250
X,0,3,0,16384 X,1,127,0,128 X,2,127,0,1
Y,3,3,0,16384 Y,4,127,0,128 Y,5,127,0,1
S,0,128,128 B,6 H,1 A;
```

Part 2 / Debugging PenPoint Applications

Chapter 4 / Introduction

The PenPoint™ operating system provides several tools that you can use to debug PenPoint programs. These include:

- A source level debugger.
- A debugging stream that can be written to a log file, viewed on a second monitor (when running PenPoint on a PC), and viewed in the PenPoint System Log application.
- Functions that write text to the debugging stream, such as **DPrintf()** and **Debugf()**.
- Macros that can write text to the debugging stream when messages return warning or error status values.
- A low-level mini-debugger.

Most of this part is dedicated to the PenPoint source level debugger, DB. The other debugging facilities are described in Chapters 10, 11, and 12 of this part.

DB Overview 4.1

DB, helps you to trace program execution and measure performance. DB has several features that allow you to interact with PenPoint at the source code and assembly language levels:

- Full support of PenPoint's multi-tasking environment. You can switch easily among different tasks to examine their states and control their execution.
- Command line evaluation of declarations and expressions. You can enter C declarations and expressions directly from the command line, allowing you to alter code for "what-if" trials.
- Batch command file and execute-on-break stored command sequences. You can build powerful debugging scripts and macros to speed up complex debugging operations.

Organization of This Part 4.2

Chapter 4, this chapter, introduces the various debugging tools and describes the rest of the part.

Chapter 5, Preparing to Run DB, describes how to compile and link your programs so that you can view symbolic names and locations in your source code while running DB.

Chapter 6, Using DB, provides a tutorial on how to use DB to perform most common debugging procedures.

Chapter 7, DB Command Reference, provides a reference to all of the DB commands.

Chapter 8, Profiling With DB, describes how to use DB's powerful profiling capabilities.

Chapter 9, Advanced DB Techniques, provides a tutorial on using some of DB's advanced features, which include: controlling threads of execution; using DB's built-in types, variables, and routines; and setting breakpoints on events.

Chapter 10, General PenPoint Debugging Techniques, describes other debugging tools and techniques.

Chapter 11, The System Log Application, describes a PenPoint application that you can use to view the debugger stream, set DEBUG flags, and view other system data.

Chapter 12, PenPoint Mini-debugger, describes the mini-debugger and the differences between it and DB.

Sample Files 4.3

This manual uses the CALC sample application in most of its examples. The source code for CALC and several other sample applications is in \PENPOINT\SDK\SAMPLE.

Chapter 5 / Preparing to Run the Debugger

Files Used in a DB Session 5.1

In addition to the program you are debugging, DB uses several other files to configure itself. Table 5-1 lists files that are involved in a debugging run. This list includes both the files that you edit and compile, and files that the PenPoint™ operating system and DB access during execution.

Table 5-1
Files Used in a DB Session

File	Usage
DB.INI	Optional file. If it exists, DB will interpret the contents of the file when it starts up.
CDB3.DLL	The user-mode portion of the PenPoint load module for DB.
CDB0.DLL	The supervisor-mode portion of the PenPoint load module for DB.
ENVIRON.INI	PenPoint reads this file at cold boot to configure the general execution environment. You can also set debugging flags in it, and specify the location of DB.INI.
*.C, *.H	The source files for your program. The Appendices contain source code for sample applications.
YOURPROG.EXE	When your compiled program has successfully linked with the PenPoint libraries the result is this PenPoint loadable module.
YOURPROG.DLL	Dynamic Link Libraries (.DLLs) are used to store shareable portions of your executable code.

Compiling and Linking 5.2

To run under DB, you follow the same compiling and linking steps covered in Chapter 6 of the *Application Writing Guide.* There are a few additional steps that you have to take to cause the compiler and linker to generate symbolic debugging information.

The WATCOM C/386 compiler and linker can generate two levels of symbolic debugging information.

- The first level of information includes line numbers and symbols for public routines. To get this level of information:
  - When compiling, set the compiler's /Of+ and /D1 switches.
  - When linking, include the line `DEBUG LINES` or `DEBUG ALL` before the `FILE` line(s) in your link command file.

- The second level of symbolic debugging information includes full symbol name and type information. To get this level of information:
  - When compiling, set the compiler's /Of+ and /D2 switches.
  - When compiling, be sure *not* to set any of the /Oalstx switches.
  - When linking, include the line `DEBUG ALL` before the `FILE` line(s) in your link command file.
- In all cases, you should set the compiler's **/en** switch if you want DB's **st** command to display routine names for modules with no symbols loaded.

You can create different levels of symbolic debugging information for different source files.

Installing DB 5.3

DB is a PenPoint DLL. To install DB, remove the "#" from the following line in \PENPOINT\BOOT\BOOT.DLC:

```
#go0-cdb0-v2(0) \\boot\386\penpoint\boot\dll\cdb0.dll
#go-cdb3-v2(0)  \\boot\386\penpoint\boot\dll\cdb3.dll
```

Installing Applications to Debug 5.4

You install the program that you wish to debug in the normal fashion. The changes to it to support debugging are in compiling and linking, not in installing.

Start PenPoint 5.5

Start PenPoint in the normal fashion.

Chapter 6 / Using DB

This chapter introduces you to DB's most important concepts and operations.

Invoking DB: the PAUSE Key 6.1

To switch from the PenPoint™ operating system UI interaction to DB interaction, press the [Pause] key on the keyboard. The current application process is frozen and the DB prompt appears.

The prompt is a greater-than character (>) followed by a tiny dot. When you type a command into DB, the command is displayed immediately to the right of the prompt:

```
>tl
```

The command is sent to DB when you press the [Enter] key.

Continuing Execution: the G Command 6.2

To continue PenPoint execution, type the **g** command.

Module Names, Process Names, and Task IDs 6.3

PenPoint executable modules (.DLLs and .EXEs) are given names when they are linked. The convention for these module names is:

```
companyName-moduleName-majorVersion(minorVersion)
```

DB gives PenPoint application processes names which are a variation on these module names. The name of a process is simply the quoted name of the module that contains the entry point for the process and followed by a number in square brackets. For instance:

```
"go-calculator-v1(0)"[1]
```

The number in square brackets indicates the process count of the application. Process count 0 is the application manager process, which owns the installed application class itself. The kernel increments an application's process count each time that application is run; process count values greater than zero indicate how many times the application has been executed (since the last cold boot).

PenPoint tasks have 16 bit task IDs. (See Chapter 75 in *Architectural Reference Manual Part 8: System Services.*)

Hexadecimal Numbers 6.4

A hexadecimal number starts with a zero (0) and uses the hexadecimal numerals 0–9 and a-f.

DB prints the leading zero in its output (except in obviously hexadecimal cases, such as dumps). DB requires that you preface all input hexadecimal numbers with a 0. 01db8 is a typical hexadecimal number.

Source Code Debugging 6.5

To debug source code, DB needs two pieces of information:

- DB needs to read symbolic debugging information. Compilers and linkers put symbolic debugging information in the .EXE or .DLL file on disk. So you may need to tell DB:
    - Where to find the .EXE or .DLL file.
    - Which .EXE or .DLL symbolic information to read.
- DB needs to know where to find the source code files.

Without this information DB can still be used, but only at the assembly code level.

The next two sections tell you how to provide this information to DB.

Finding and Loading Symbols: the SYM Command 6.5.1

Use the **sym** command to tell DB to read symbolic debugging information into memory:

```
>sym "calculator"
>sym "calc_eng"
```

If the module doesn't contain symbolic debugging information (probably because it was compiled or linked without the appropriate flags), DB displays a warning.

See Section 6.16.3 for information on **sym** command options that can reduce the amount of memory required.

Using Source Code: the SRCDIR Command 6.5.2

To display source code and to allow DB to display source code (e.g. after taking a breakpoint) DB must be told where source files are located.

DB's **srcdir** command tells DB where to find the source files that correspond to a given module. The volume name (after the \\) must be the DOS volume name. You can see the DOS volume name with the DOS DIR or CHKDSK command; you can modify it with the DOS LABEL command.

```
>srcdir "calculator" \\c\penpoint\sdk\sample\calc
>srcdir "calc_eng" \\c\penpoint\sdk\sample\calc
```

Alternatively, you can set the DBSrc environment variable in \PENPOINT\BOOT\ENVIRON.INI. This is particularly useful if you commonly work with source files in many directories. DB will search these directories *after* searching any directories specified in **srcdir** commands.

```
DBSrc=\\c\dir1;\\c\dir2;\\c\dir3
```

Setting DB's Context: the CTX Command 6.6

DB is always focused on some area of the program that you are debugging. This focus is called the **context**. The context includes:

- A task ID (the current task) and its current code addresss.
- A stack frame of a currently active function call (the current call).
- A name scope in which to look up identifiers (the current scope).

The **ctx** command sets DB's context. DB suspends the task and sets the current call and current scope, based on the task's current code address.

To set the context to a particular task, type the following. (DB's **tl** command lists all tasks.) When you set DB's context to a given task, DB's prompt changes to include the process name and ID of that task. In this example, *"go-calculator-v1(0)"[1]* is DB's name for the process, and *0508* is PenPoint's task ID.

```
>0508 ctx
"go-calculator-v1(0)"[1] 0508>
```

To set the context to the executable module for the Calculator, enter the following. DB will be set to the context of the CALC.EXE module. This makes the set of identifiers in the module the currently available ids.

```
>ctx "calc.exe"
```

To see the current lexical scope, enter the ctx command with no parameters.

Breakpoints: the BP, BL, and BC Commands 6.7

Setting a breakpoint at an instruction halts execution when that instruction is hit. Then you can examine the variables and CPU registers to see what your program has been doing up to that point.

With PenPoint's multitasking environment, multiple tasks may simultaneously execute the same code. By default, DB sets breakpoints so that they affect all tasks, but you can restrict a breakpoint so that it only affects a specified set of tasks.

The **bp** command sets a breakpoint. This example sets a breakpoint at the entry point of CalcEngineEnterOperator:

```
>bp CalcEngineEnterOperator
```

The **bl** command lists all breakpoints. The listing shows which tasks will be trapped at the breakpoint (in the square brackets), and then shows the location of the breakpoint.

```
>bl
  1: [*] BP CalcEngineEnterOperator.@381
```

After setting a breakpoint, enter the **g** command to resume execution.

```
>g
```

When a breakpoint is hit, DB suspends the task and displays the following information. The first line shows which task hit the breakpoint. The second line displays the source code line on which the breakpoint was set. This is the instruction that is about to be executed.

```
00508 P "go-calculator-v1(0)"[1]                     1.272 1: BP
 381>>     P_CALC_ENGINE_DATA  pInst = *pData; // Reduce dereferences.
```

You can also set a breakpoint at a specific line of source code, as follows. If you do not qualify the line number with a file name or a routine name, DB will try to use the file containing the current source line.

```
"go-calculator-v1(0)"[1] 0508>bp @381
"go-calculator-v1(0)"[1] 0508>bp "calceng.c".@381
```

A breakpoint can be cleared with the **bc** command.

Viewing the Call Stack: the ST Command 6.8

You can examine a task's call stack with the **st** command. The call stack maintains local variables and parameters as your program moves from one scope to another.

```
"go-calculator-v1(0)"[1] 0508>st
%1 >
CalcEngineEnterOperator(
    msg: 25170108, self: 01ad103fb, pArgs: 043444c, ctx: 0432be8,
    pData: 043fe93cc)
%2   ObjectCall [ msgCalcEngineEnterOperator [dyn 1ad103fb] 043444c 0 ]
%3
CalcEngineProcessKey(
    msg: 8392892, self: 01ad103fb, pArgs: 043444c, ctx: 0432c48,
    pData: 043fe93cc)
%4   ObjectCall [ msgCalcEngineProcessKey [dyn 1ad103fb] 043444c 043a478d5 ]
%5
CalcAppButtonNotify(
    msg: 117440616, self: 01ac403e5, pArgs: 05, ctx: 0432ca0, pData: 043fe8cb2)
%6   ObjectCall [ msgButtonNotify [dyn 1ac403e5] 05 01ad90402 ]
%7   ButtonNotifyClient [ 01ad90402 043fe99b0 0432d98 0409bc610 ]
%8   CallMethod [ 07000068 01ad90402 0432d98 0432d60 ]
...
```

The sequence begins with the top of the stack and then lists the stack back to its base. Repeated **st** commands show more of the stack.

The names of functions are displayed along with the values of the parameters that were passed to them at the time of the call (shown in parentheses). Local variables are listed in the line above the function description, indented and surrounded with curly braces.

The hexadecimal numbers displayed between square brackets (in frames %4, %6, %7, and %8) are the values of variables of functions in modules for which there is no symbolic debugging information.

Frame Numbers and the CTX Command 6.8.1

You can use the frame numbers displayed by the **st** command as parameters to the **ctx** command.

```
"go-calculator-v1(0)"[1] 0548>ctx %3
"go-calculator-v1(0)"[1] 0548>ctx
%3   "calc.dll"."calceng.obj".CalcEngineProcessKey."calceng.c".@482
```

You can use the ctx top command to get back to the top of the call stack.

Examining and Setting Values 6.9

DB has several commands to examine and set variables, and to evaluate expressions.

The ? (Evaluate) Command 6.9.1

DB's ? ("evaluate") command evaluates a C expression and prints the resulting value.

```
"go-calculator-v1(0)"[1] 0508>?pArgs->buf
043444e
```

The ? command can display values in several different formats.

```
"go-calculator-v1(0)"[1] 0508>?pArgs->buf
043444e
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,s
"6"
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,x
043444e
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,d
4408398
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0]
54
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'6'
```

The ? command can be used to structures as well as variables and constants.

```
"go-calculator-v1(0)"[1] 0508>?pData
043fe93cc
"go-calculator-v1(0)"[1] 0508>?*pData
043474c
"go-calculator-v1(0)"[1] 0508>?**pData
 {
    xValue: 1.125000; yValue: 77.000000; pendingOp: 5;
    keysSeen: {0, 55, 0, 0, 0,... 0, 0, 0, 0, 0}; numberEntered: 1;
    calcError: 0;}
```

In this example, DB displays the address for the entry point of a function.

```
>? CalcEngineEnterOperator
043a1c7b8
```

The ? command can also set values. Simply put the variable on the left-hand side of a replacement operator (=) and supply a valid expression on the right-hand side.his example changes the first element in the buffer from the value of "6" to the value of "7."

```
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'6'
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0] = '7'
55
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'7'
```

Known Identifiers: the IDS command 6.9.2

The **ids** command lists the identifiers that are "known" or visible in the current context. These identifiers can be used to set breakpoints, change the value of variables under DB control, and pinpoint locations in the source code that you would like to view.

```
"go-calculator-v1(0)"[1] 0508>ctx %1
"go-calculator-v1(0)"[1] 0508>ids
{ (vars) pInst result s}
{ (params) msg self pArgs ctx pData}
{}
{ (tags) CALC_ENGINE_DATA CALC_ENGINE_NEW CALC_ENGINE_TOKEN OBJ_RESTORE
OBJ_SAVE OBJECT_NEW (vars) clsCalcEngineTable (functions) _8087
CalcEngineAppendCharToToken CalcEngineDump CalcEngineEnterNumber
CalcEngineEnterOperator CalcEngineEvalBinary CalcEngineEvalUnary
CalcEngineFree CalcEngineInit CalcEngineProcessKey CalcEngineRestore
CalcEngineSave CalcEngineSaveValue CalcEngineUpdateToken DllMain}
"go-calculator-v1(0)"[1] 0548>ids %3
{ (vars) s}
{ (params) msg self pArgs ctx pData}
```

Identifier Types: the TYPE Command 6.9.3

DB's **type** command displays the type of its parameter. If the parameter is a structure, **type** also displays the type of each field.

```
"go-calculator-v1(0)"[1] 0508>type self
ptr to void
"go-calculator-v1(0)"[1] 0508>type pArgs
ptr to struct CALC_ENGINE_TOKEN
"go-calculator-v1(0)"[1] 0508>type pData
ptr to ptr to struct CALC_ENGINE_DATA
"go-calculator-v1(0)"[1] 0508>type *pData
ptr to struct CALC_ENGINE_DATA
"go-calculator-v1(0)"[1] 0508>type **pData
struct CALC_ENGINE_DATA {
    xValue: double; yValue: double; pendingOp: short; keysSeen:
    array [30] of unsigned char; numberEntered: short; calcError: short;}
```

In this example, **type** displays the type of a function:

```
>type CalcEngineEnterOperator
function(
    msg: long, self: ptr to void, pArgs: ptr to struct CALC_ENGINE_TOKEN, ctx:
    ptr to void, pData: ptr to ptr to struct CALC_ENGINE_DATA, ...)
returning long
```

Lexical Scope 6.9.4

The "scope" of identifiers follow the conventions of the C programming language. While in the context of a function, you cannot directly examine the state of variables in another function. Similarly, variables defined within a program block (surrounded by curly braces) are not visible outside of that block.

You can use full **Scope.Identifier** specification to examine the state of variables that are outside the current DB context. This is discussed in Section 7.2.6.

Single-Stepping: the p, P, t, and T Commands 6.10

DB provides four commands that allow you to single-step the execution of a program on a statement-by-statement basis:

- p Single-steps through *source* statements, *passing over* called routines.
- P Single-steps through assembly instructions, *passing over* called routines.
- t Single-steps through *source* statements, *stepping into* called routines.
- T Single-steps through assembly instructions, *stepping into* called routines.

Viewing Source Code: the V Command 6.11

The v command displays the source code of a program. You can display the source line that is just about to be executed. You can move around in the source files using function declarators, memory addresses, and line numbers to specify which section of the source code to display.

Before viewing source code, you must tell DB which directory contains the source code files. Use either the **srcdir** command or the DBSrc environment variable. (See the description of the **srcdir** command in Chapter 7.)

With the source directory specified, you are ready to display the source code for your program.

This example displays the source code of the beginning of the routine CalcEngineEnterOperator.

```
>v CalcEngineEnterOperator
  378   MsgHandlerWithTypes(CalcEngineEnterOperator, P_CALC_ENGINE_TOKEN, \
  379           PP_CALC_ENGINE_DATA)
  380   {
  381      P_CALC_ENGINE_DATA  pInst = *pData; // Reduce dereferences
  382       double               result;
  383       STATUS               s;
  384
  385       if (pArgs->key == nop) {
```

You can also directly specify an instruction memory address to the v command. (See Chapter 7 for a description of the ? command used in this example.)

```
>? CalcEngineEnterOperator
043a1c7b8
>v 043a1c7b8
 376   */
 377
 378   MsgHandlerWithTypes(CalcEngineEnterOperator, P_CALC_ENGINE_TOKEN, \
 379          PP_CALC_ENGINE_DATA)
 380   {
 381       P_CALC_ENGINE_DATA  pInst = *pData; // Reduce dereferences
 382       double              result;
 383       STATUS              s;
```

You can also specify a line number to be viewed. (By default the **v** command displays lines on either side of the line number requested. See the Datasheet for complete information.)

```
"go-calculator-v1(0)"[1] 0508>v @204F255D
 201   {
 202       U16                     len;
 203
 204      len = strlen(pInst->keysSeen);
 205
 206       if (len >= maxDigits-1) {
 207           return;
 208       }
```

Whenever DB returns from executing statements, you can display the code that is pending execution with the **v** and no parameters:

```
"go-calculator-v1(0)"[1] 0508>v
 378   MsgHandlerWithTypes(CalcEngineEnterOperator, P_CALC_ENGINE_TOKEN, \
 379          PP_CALC_ENGINE_DATA)
 380   {
 377>>     P_CALC_ENGINE_DATA  pInst = *pData; // Reduce dereferences
 381       double              result;
 382       STATUS              s;
 383
 384       if (pArgs->key == nop) {
```

Repeating the **v** command displays the next several lines of source code. In this way you can page down through a source file.

Viewing Assembly Code: the U Command 6.12

The **u** command displays the assembly code of a program. Variables are displayed as addresses and constants are displayed as the values themselves. The following command displays assembly code starting at the address indicated by the identifier CalcEngineEnterOperator.

```
>u CalcEngineEnterOperator
43A1C7B8  55                   PUSH  EBP
43A1C7B9  89 E5                MOV   EBP, ESP
43A1C7BB  83 EC 10             SUB   ESP, 16
43A1C7BE  8B 45 18             MOV   EAX, DWORD PTR [EBP+24{pData}]
43A1C7C1  8B 00                MOV   EAX, DWORD PTR [EAX]
43A1C7C3  89 45 FC             MOV   DWORD PTR [EBP-4{pInst}], EAX
43A1C7C6  8B 45 10             MOV   EAX, DWORD PTR [EBP+16{pArgs}]
43A1C7C9  66 83 38 00          CMP   WORD PTR [EAX], 0
43A1C7CD  75 05                JNZ   043A1C7D4{CalcEngineEnterOperator.@385}
43A1C7CF  E9 AD 01 00 00       JMP   043A1C981{CalcEngineEnterOperator.@426}
```

If you do not specify an address, DB starts from where it left off in the previous **u** command.

You can also specify an address directly:

```
>? CalcEngineEnterOperator
043a1c7b8
>u 043a1c7b8
43A1C7B8   55                 PUSH  EBP
43A1C7B9   89 E5              MOV   EBP, ESP
43A1C7BB   83 EC 10           SUB   ESP, 16
43A1C7BE   8B 45 18           MOV   EAX, DWORD PTR [EBP+24{pData}]
43A1C7C1   8B 00              MOV   EAX, DWORD PTR [EAX]
43A1C7C3   89 45 FC           MOV   DWORD PTR [EBP-4{pInst}], EAX
43A1C7C6   8B 45 10           MOV   EAX, DWORD PTR [EBP+16{pArgs}]
43A1C7C9   66 83 38 00        CMP   WORD PTR [EAX], 0
43A1C7CD   75 05              JNZ   043A1C7D4{CalcEngineEnterOperator.@385}
43A1C7CF   E9 AD 01 00 00     JMP   043A1C981{CalcEngineEnterOperator.@426}
```

Executing C Code 6.13

DB includes a C language interpreter which can interactively execute code during a debugging session.

With the interpreter, DB can also use C expressions to represent addresses and variables.

When interpreting DB commands, the ! command causes DB to switch to interpreting C code. Conversely, when interpreting C code, the ! command causes the C interpreter to interpret the next statement as a DB command.

Executing Code at the DB Prompt 6.13.1

This example executes C code which changes an array of values. The ! command is used to enter C declarations and statements.

```
"go-calculator-v1(0)"[1] 0508>!int i;
"go-calculator-v1(0)"[1] 0508>!for (i=0; i<4; i++) (*pData)->keysSeen[i] = '7';
"go-calculator-v1(0)"[1] 0508>? (*pData)->keysSeen,s
"7777"
```

Executing Code at a Breakpoint 6.13.2

You can specify code to be executed when a breakpoint is hit.

```
>bp CalcEngineEvalBinary {
}  printf("op=%d opnd2=%f\n",op,opnd2);
}  if ((op==5) && (opnd2==0.0)) {
}    printf("got divide by zero!\n");
}    !break;
}  }
}}
>g
```

Notice that the **break** command in the if-statement is prefixed with the ! command; this tells the C interpreter that the following command is a DB command rather than C code.

Also notice that while we're entering C code into a block delimited by braces ({ and }), DB's prompt changes from a greater-than sign (>) to a brace ({).

Now when the Calculator's buttons are tapped, the breakpoint is hit and DB executes the code inside the curly braces of the **bp** command. The printf statement prints out the current value for some variables. The **break** command returns control to the DB prompt.

```
op=5 opnd2=9.000000
op=2 opnd2=6.000000
op=4 opnd2=3.000000
op=3 opnd2=9.000000
op=5 opnd2=.000000
got divide by zero!
 00508 P "go-calculator-v1(0)"[1]                        3.199 1: BP
   80>>     STATUS               s = stsOK;
>
```

Task List: the TL Command 6.14

To see the list of all tasks, issue this command:

```
>tl
 00508 P "go-calculator-v1(0)"[1]                          .744 Msg
 004e8 P "go-nbtoc-v1"[1]                                  .909 Msg
 004d8 P "go-nbapp-v1"[1]                                  .942 Msg
 004c8 P "go-dtapp-v1"[1]                                 1.129 Msg
 004b8 P "go-calculator-v1(0)"[0]                          .155 Msg
 004a8 P "go-calc_eng-v1(0)"[0]                             .007 Msg
 ...
 002e8 P "go-settings_init-v1"[0]                          .116 Msg
 002a8 P "go-settings-v1"[0]                               .295 Msg
 00288 P "go-minitext-v1"[0]                               .373 Msg
 00268 P "go-notepaperapp-v1"[0]                           .200 Msg
 00258 P "go-notepaper-v1(0)"[0]                           .041 Msg
 00238 P "go-nbapp-v1"[0]                                  .125 Msg
 00218 P "go-nbtoc-v1"[0]                                  .127 Msg
 001f8 P "go-sectapp-v1"[0]                                .169 Msg
```

The process names that are followed by a zero in square brackets ([0]) are the application monitor processes for the installed application. The process names that are followed by numbers greater than zero in square brackets (such as [1] and [2]) are the processes associated with active documents.

Saving Typing 6.15

DB provides several ways to save typing common commands.

DB.INI File 6.15.1

At system start-up time, DB looks for a startup file; if it finds one, it reads commands from the file. You can use the startup file to store a startup sequence for your debugging session, accelerating the debugging part of a edit-compile-debug cycle.

DB first looks for the DBINI environment variable in \PENPOINT\BOOT\ENVIRON.INI:

```
DBINI=\\c\mydir\mydb.ini
```

If DBINI is not defined, DB looks for \PENPOINT\BOOT\DB.INI and runs it if it exists. If DB doesn't find a DB.INI, it returns control to PenPoint (it acts as if a user typed a "g" command).

Using DB Scripts 6.15.2

You can use the < command to read a file that contains DB commands. For example, you could create a DOS file \GO\CALC\SCRIPT.DB that contains the following lines:

```
sym "calculator"
sym "calc_eng"
srcdir "calc_eng" \\c\penpoint\sdk\sample\calc
srcdir "calculator" \\c\penpoint\sdk\sample\calc
```

Then all you have to do is enter the following command when you want to link the symbolic debugging information into your DB session:

```
> < \\c\go\calc\script.db
```

Command Line Editing 6.15.3

You can edit DB command lines with these keys: [←], [→], [Home], [End], [Del], and [Bksp]. You can use the [Esc] key to cancel the entire command. The "cursor" is shown as a tiny dot.

The up and down arrow keys recall previous commands, which can be edited, if necessary.

DB and Memory Use 6.16

The symbolic debugging information used by DB consumes memory. Combined with the overhead for DB itself, your program may run into memory constraints.

Checking Available Memory: the MI Command 6.16.1

You can check on available memory with DB's **mi** command.

Eliminating Applications 6.16.2

To conserve memory, you can boot a reduced PenPoint configuration by removing unneeded applications from \PENPOINT\BOOT\APP.INI. Once PenPoint is running, you can delete documents and de-install or deactivate applications in the usual manner.

Loading Partial Symbolic Debugging Information 6.16.3

Using the compiler's /D2 switch and the linker's DEBUG ALL switch produces a lot of debugging information; DB may not have enough memory to read it all. To get around this, the **sym** command has an optional form.

The following command will load full symbolic debugging information for obj1.obj and obj2.obj, but only line numbers and global identifiers for the rest of the module's object files.

```
>sym "myapp" (obj1,obj2)
```

The following command, with no files listed between the parentheses, loads only line numbers and global identifiers for all object files in the module:

```
>sym "myapp" ()
```

See the Datasheet for the **sym** command for details.

Using DB to Send Messages 6.17

You can ask DB to have current task execute **ObjectCall**, **ObjectCallAncestor**, **ObjectSend**, **ObjectSendUpdate**, **ObjectPost** or **ObjectPostAsync** by using those functions in a DB ? command. For instance:

```
> ?ObjectCall(msg, object, 0)
```

String Names for Messages, Objects, and Statuses 6.18

You can "encode" a PenPoint message, object/class, or status name within any DB expression with the construct

```
[msg|obj|sts <name>]
```

For instance, the following will send msgDump to theProcess.

```
> ?ObjectCall([msg msgDump], [obj theProcess], 0)
```

If you have loaded the class manager symbols, DB can interpret PenPoint message, object, class, and status identifiers. Thus you could enter the previous command as:

```
> ?ObjectCall(msgDump, theProcess, 0)
```

Watching Memory: the ON ACCESS and ON STORE Commands 6.19

You can use DB's **on** command to interrupt execution when memory locations are read or written. There are two variations that are useful for watching memory:

There are several other variations of the **on** *command. See Section 9.5 for more information.*

`on store lvalue` This variation breaks to DB when the memory specified by lvalue is written.

`on access lvalue` This variation breaks to DB when the memory specified by lvalue is read or written.

An lvalue is typically a variable name, although it can also be any expression referencing a region of storage. For a complete definition of lvalues, see any good reference book on the C programming language, such as *The C Programming Language* by Kernighan and Ritchie.

In the following example, assume that you have taken a breakpoint at the routine CalcEngineEnterOperator and that you decide that you need to break whenever the value of (*pData)->numberEntered changed.

```
"go-calculator-v1(0)"[1] 0508>?**pData
 {
   xValue: 3.092736e-14; yValue: 3.092736e-14; pendingOp: 5;
   keysSeen: {0, 55, 0, 55, 56,... 0, 0, 0, 0, 0}; numberEntered: 0;
   calcError: 0;}
"go-calculator-v1(0)"[1] 0508>on store (*pData)->numberEntered
"go-calculator-v1(0)"[1] 0508>bl
  1: [*] on Store [target task 0508 logAddr 43477C] 1 2 { }
```

Execution will halt immediately *after* (*pData)->numberEntered has been written; you may have to give the **v b** command to display the previous source code.

```
 00508 P "go-calculator-v1(0)"[1]                    1.948 1: Store
  430>>     return stsOK;
"go-calculator-v1(0)"[1] 0508>v
  427
  428       }
  429
  430>>     return stsOK;
  431       MsgHandlerParametersNoWarning;
  432
  433   } /* CalcEngineEnterOperator */
  434
"go-calculator-v1(0)"[1] 0508>v b
  419                   Dbg(if (s != stsCalcEngineComputeError) StsWarn(s);)
  420                   return s;
  421               }
  422               pInst->pendingOp = nop;
  423               CalcEngineSaveValue(result, pArgs, pInst);
  424           }
  425           pInst->pendingOp = pArgs->key;
  426          pInst->numberEntered = false;
"go-calculator-v1(0)"[1] 0508>g
 00508 P "go-calculator-v1(0)"[1]                    2.106 1: Store
  366>>     return stsOK;
"go-calculator-v1(0)"[1] 0508>v
  363       CalcEngineSaveValue(pArgs->value, pArgs, *pData);
  364       (*pData)->numberEntered = true;
  365
  366>>     return stsOK;
  367       MsgHandlerParametersNoWarning;
  368
  369   } /* CalcEngineEnterNumber */
  370
```

If you want to break whenever memory is either read or written, use **on access**.

```
"go-calculator-v1(0)"[1] 0508>on access (*pData)->numberEntered
"go-calculator-v1(0)"[1] 0508>bl
  1: [*] on Access [target task 0508 logAddr 43477C] 1 2 { }
```

Now execution will halt whenever the location is read or written. For example, notice that the calculator application now stops at line 416:

```
"go-calculator-v1(0)"[1] 0508>g
 00508 P "go-calculator-v1(0)"[1]                        2.302 1: Access
  416>>         if ((pInst->pendingOp != nop) AND (pInst->numberEntered)) {
```

These two commands can operate on raw memory addresses as well as on variables. As examples, you may want to use a raw address if you need to watch allocated memory or if you're debugging a module without symbolic debugging information. In the following example, we'll compute the address to watch.

```
"go-calculator-v1(0)"[1] 0508>? &((*pData)->numberEntered)
043477c
"go-calculator-v1(0)"[1] 0508>type (*pData)->numberEntered
short
"go-calculator-v1(0)"[1] 0508>on access *(short *)043477c
```

Events that Activate DB 6.20

There are three circumstances which lead to the DB prompt appearing:

- If the user types DB's Pause key.
- If any task experiences a fault, such as a page fault or divide by zero.
- If any code calls the Debugger PenPoint function.

Exiting DB and PenPoint 6.21

To exit DB and PenPoint, enter the **q** command at the DB prompt:

```
>q
```

Chapter 7 / DB Command Reference

This chapter describes the commands that you can enter at the DB prompt.

Most of the command descriptions are accompanied by examples of command usage. Be aware that the line numbers that appear in these examples might not match line numbers that you see while examining the same sample application.

Command Summary 7.1

Table 7-1 lists many of DB's commands in functional categories.

Table 7-1
DB Command Summary

Command	Description
	Process and Task Commands
ctx	Selects the current task and sets DB's lexical scope.
tl	Display PenPoint's task list.
ti	Display information about a task.
	Execution Control Commands
g	Resumes PenPoint execution.
P	Single-steps assembly instructions, passing over called routines.
p	Single-steps source statements, passing over called routines.
T	Single-steps assembly instructions, stepping into called routines.
t	Single-steps source statements, stepping into called routines.
q	Quits DB.
	Breakpoint Commands
bp	Sets a breakpoint.
bl	Lists breakpoints.
bc	Clears the specified breakpoint.
on	Creates a "handler" for specified events.
bd	Disables the specified breakpoint.
be	Enables the specified breakpoint.
	Display Commands
!	Switches the interpreter from DB commands to C code, or vice-versa.
?	Evaluates and/or assigns a C expression.
d (db, dw, dd)	Displays memory contents.
u	Displays assembly code.
uv	Displays assembly code with interspersed source code.

continued

Table 7-1 (continued)

Command	Description
v	Displays source code.
vu	Displays source code with interspersed assembly language statements.
st	Display the call stack.
r	Displays the registers.
ai	Displays address information.
ids	Displays the identifiers known in a scope.
fns	Displays the functions known in a scope.
vars	Displays variables known in a scope.
id	Displays the type and declaration information for an identifier.
type	Displays type that a C expression evaluates to.
cm, co, cs	Converts messages, objects, and statuses into their string formats.
	File Commands
srcdir	Tells DB where to find source files for a module.
sym	Loads symbolic debugging information into memory.
<	Reads a file as input for DB.
files	Displays the files associated with a scope.
	Profiling Commands
profile	Creates a profile breakpoint.
dp	Displays profile data.
zp	Clears profile data.
	Miscellaneous Commands
fl	Lists values of debugging flags.
fs	Sets values of debugging flags.
ver	Displays DB's version.
h	Displays help on DB commands and topics.
log	Starts and stops logging.
mi	Displays memory information.
mini	Enters PenPoint's mini-debugger.
od	Sends msgDump to an object.
break	Returns control to the DB prompt.

Notation Conventions 7.2

This chapter uses the notation conventions described in the next few sections.

Scope Specification (*scopeSpec*) 7.2.1

A *scope specification* (or *scopeSpec*) specifies a position in a task's stack. Changing the scope shifts the lexical scope for DB within the task. When examining variables, you will see values for the current scope.

A *scopeSpec* takes the following forms:

```
[ {into | outto} . ] {  functionName     |
                        filename         |
                        programName      |
                        scopeName        |
                        @ lineNumber     |
                        % stackFrameNumber}
```

or a string of the following elements separated by periods (to cause multiple moves into or out of a scope level):

```
top | in | out | inb | outb
```

Table 7-2
Scope Specification

Keyword	Usage
into	Causes DB to move in to the scope of the function or line number after the dot.
outto	Causes DB to move out to the scope of the function or line number after the dot.
functionName	A function available in the current scope.
filename	A file name established with the srcdir and v commands, or the name of a loaded module (with file extension).
programName	The name of a load module. (These names are listed by the sym command.)
scopeName	The name of a scope.
%stack frame number	The number for a scope shown by the st command.
@lineNumber	A line number in the source file associated with the current scope. The "@" is required. (The v command shows the assignment of line numbers to source files.)
top	Positions DB at the top of a task's call stack (the innermost scope).
in	Move "in" one CALL on the stack.
out	Move "out" one CALL on the stack.
inb	Move in one lexical block.
outb	Move out one lexical block.

For example, a *scopeSpec* is supplied to the ctx command (described elsewhere). The following commands change the scope of the current task:

```
"go-calculator-v1(0)"[1] 0508>ctx outto.CalcEngineProcessKey
"go-calculator-v1(0)"[1] 0508>ctx
%4   "calc.dll"."calceng.obj".CalcEngineProcessKey."calceng.c".@478
"go-calculator-v1(0)"[1] 0508>ctx in
"go-calculator-v1(0)"[1] 0508>ctx
%3   go-kernel3-V0(1).01:06bea
"go-calculator-v1(0)"[1] 0508>ctx out
"go-calculator-v1(0)"[1] 0508>ctx
%4   "calc.dll"."calceng.obj".CalcEngineProcessKey."calceng.c".@478
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>ctx
%1   "calc.dll"."calceng.obj".CalcEngineEvalBinary."calceng.c".@76
```

Line Count (*lineCount*) 7.2.2

Several DB commands (for instance, **v** and **u**) take an optional parameter known as a *lineCount*. The line count controls how many lines are displayed, and where the display of lines starts.

```
[ a | b | l ] [count]
```

count The number of lines to be displayed.

a "Around." The command displays 1/2 of *count* lines before the target and 1/2 after.

b "Before." The command displays *count* lines up to and including the target.

l "List." The command displays *count* lines starting at the target.

Code Addresses 7.2.3

A *codeAddress* is:

- A C function identifier.
- A line number.
- A literal address specification.
- Any C expression that yields a function pointer..

For example, the **g** command can optionally set a temporary breakpoint at the code address of a routine and then let execution proceed up to that point:

```
>g CalcEngineEvalBinary
```

Data Addresses 7.2.4

A *dataAddress* is:

- A C pointer variable identifier.
- A literal address specification.
- A C expression that yields a pointer.

Line Numbers 7.2.5

A line number is expressed by the `@` sign followed by a decimal literal. If the line number is not qualified, DB takes the line to be in the same file as the current source position.

You can qualify the line number with a *scopeSpec*, for example `"calc.c".@104` or `SomeRoutine.@104`.

"Scope.Identifier" Referencing 7.2.6

An identifier outside of the current scope can be referenced with a *Scope.Identifier* reference. The reference consists of a *scopeSpec* followed by a period followed by an identifier. For example:

```
"calceng"."calceng.obj".CalcEngineEvalBinary
```

references the function CalcEngineEvalBinary as it occurs in the scope of its load module "calc_eng." This reference can occur anywhere that a standard C identifier can.

Task Set (*taskSet*) 7.2.7

Many DB commands operate on a set of tasks, or ***taskSet***. A task set is one or more task IDs. If necessary, a task set can be syntactically distinguished from a command by enclosing the task set in square brackets.

There are several ways to specify a task set; most DB users will only need the first two or three forms.

Table 7-3
Specifying a Task Set

Entry	Explanation
taskId	a single task
*	all tasks
.	the current task
taskId *	all the tasks in the specified task's parent process
taskSet +/– taskSet	union or difference of two taskSets
"progName"	the program name bound to a .EXE or .DLL module, as listed by the tl command
"progName"[instanceNumber]	an instance of an .EXE as listed by tl
"#taskName"	a task name, as set by the OSTaskNameSet() function

Command Datasheets 7.3

This section contains a complete reference data sheet for each of the DB commands. Each datasheet explains all the arguments in a command, and examples of usage are given. The commands are listed alphabetically.

!

Switches the interpreter from DB commands to C code, or vice-versa.

Syntax

! *C code*

C code Any legal C declaration or statement.

Examples

```
> !typedef int LENGTH;
> !LENGTH maxlen;
> !maxlen = 2;
```

Remarks

The ! operator can be used to declare new data.

The ! operator can also be used to escape to a DB command where DB is expecting a C statement. --Refer to Section 6.13.

<

Reads a file as input for DB. The file should contain DB commands or C code.

Syntax

< ***filename***

filename A PenPoint file name.

Examples

```
> < script.db
```

Remarks

If the filename ends in ".c" the file is treated as C code; otherwise it is treated as a collection of DB commands.

?

Evaluates a C expression and prints the resulting value. It can also assign a value to a variable.

Syntax

? ***C-expression*** [***,fmt***]
? ***C-expression*** **=** ***C-expression***

C-expression Any expression that can be evaluated within the current context. Identifiers that are outside the current scope can be used, but must be qualified as described in Section 7.2.6. Identifiers from another task cannot be referenced. Identifiers from an inner scope cannot be referenced.

fmt Optionally tells DB how to format the value. When *fmt* is used, the comma is required. If the value of the expression is a struct, a string of format characters may be specified, one for each element. These formats are allowed:

d displays a decimal value

x displays a hex value

p displays a pointer

s displays a string

c displays a character

Examples

```
"go-calculator-v1(0)"[1] 0508>?pArgs->buf
043444e
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,s
"6"
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,x
043444e
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,d
4408398
"go-calculator-v1(0)"[1] 0508>?pArgs->buf,s
"6"

"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0]
54
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'6'
```

```
"go-calculator-v1(0)"[1] 0508>?pData
043fe93cc
"go-calculator-v1(0)"[1] 0508>?*pData
043474c
"go-calculator-v1(0)"[1] 0508>?**pData
 {
    xValue: 1.125000; yValue: 77.000000; pendingOp: 5;
    keysSeen: {0, 55, 0, 0, 0,... 0, 0, 0, 0, 0}; numberEntered: 1;
    calcError: 0;}
"go-calculator-v1(0)"[1] 0508>? CalcEngineEnterOperator
043a1c7b8
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'6'
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0] = '7'
55
"go-calculator-v1(0)"[1] 0508>?pArgs->buf[0],c
'7'
```

Remarks

If the value being printed by ? is a structure, *fmt* can be a string to specify formats for the structure members.

---

ai

Displays address information.

Syntax

ai *dataAddress*

dataAddress A data address as described in Section 7.2.4.

Examples

```
"go-calculator-v1(0)"[1] 0508>ai pArgs
task 0508 000434000..000437fff user private rw offset 044c
```

---

bgNc

Clears the specified breakpoint.

Syntax

bc { *breakpoint* | * }

breakpoint DB's breakpoint number as displayed by the **bl** command.

* all breakpoints.

Examples

Clears breakpoint 2.

>bc 2

Clears all breakpoints.

>bc *

Remarks

The **bc** command permanently clears a breakpoint; see the **bd** command for a discussion of how to temporarily disable breakpoints.

bd

Temporarily disables the specified breakpoint.

Syntax

bd { *breakpoint* | * }

breakpoint DB's breakpoint number as displayed by the **bl** command.

* all breakpoints.

Examples

In the following example, notice how the disabled breakpoint is displayed by the **bl** command.

```
>bp CalcEngineEvalBinary
>bp CalcEngineEnterNumber
>bd 1
>bl
  1: [*] BP (disabled) CalcEngineEvalBinary.@76
  2: [*] BP CalcEngineEnterNumber.@359
```

Remarks

The **bd** command temporarily disables one or all breakpoints. This is particularly useful if you've set up complex breakpoints that you'd like to temporarily disable without having to re-enter them again later.

If ***breakpoint*** is a profile breakpoint, **bd** stops accumulating execution data, but retains its current data. See the **zp** command for information about clearing data.

See the **bc** command for a discussion of how to permanently remove breakpoints.

See the **be** command for a discussion of how to re-enable breakpoints.

be

Enables the specified breakpoint.

Syntax

be { *breakpoint* | * }

breakpoint DB's breakpoint number as displayed by the **bl** command.

* all breakpoints.

Remarks

See the bd command for a discussion of how to disable breakpoints.

bl

Lists breakpoints.

Syntax

bl

Examples

```
>bp CalcEngineEvalBinary
>. bp CalcEngineEnterNumber
>profile +c CalcEngineEvalUnary
>bl
  1: [*] BP CalcEngineEvalBinary.@80
  2: [0508] BP CalcEngineEnterNumber.@363
  3: [*] Profile +c  go-calc_eng-V1(0) 1.2aa:call
```

Remarks

The **bl** command lists breakpoints set with the **bp** or **profile** commands.

Technically, **bl** lists all event handlers. Breakpoints and profiles are by far the most common event handlers. See the discussions of the **on**, **on access** and **on store** commands for descriptions of other event handlers.

bp

Sets a breakpoint.

Syntax

[*taskSet*] **bp** *codeAddress* [*eventAction*]

taskSet A task set as described in Section 7.2.7. Defaults to all tasks.

codeAddress The address for the breakpoint as described in Section 7.2.3.

eventAction C code to be executed when the breakpoint is hit.

Examples

This command sets a breakpoint at the entry point of a function for all tasks.

```
>bp CalcEngineEvalBinary
```

Sets a breakpoint at the entry point of a function in a single task.

```
>0508 bp CalcEngineEvalBinary
```

Sets a breakpoint at the instruction at specific code address for all tasks.

```
>bp 043a1c195
```

Sets a breakpoint at line 377 of the current source context.

```
>bp @377
```

Sets a breakpoint at the entry to CalcEngineEvalBinary. Whenever this breakpoint is hit, the contents of the variables **op** and **opnd2** are printed. Under the right conditions, control is returned to DB; otherwise the program simply continues execution.

```
>bp CalcEngineEvalBinary {
}  printf("op=%d opnd2=%f\n",op,opnd2);
}  if ((op==5) && (opnd2==0.0)) {
}    printf("got divide by zero!\n");
}    !break;
}  }
}}
```

break

From C code, returns control to the DB prompt.

Syntax

break

Examples

Normally, breakpoints with attached code do not pause; they simply continue program execution. If you want to attach code but have the breakpoint stop program execution, use the **break** command in the attached code.

```
>bp SomeRoutine {printf("%s\n", string); !break;}
```

cm, co, cs

Converts messages, objects, and statuses into their string formats.

Syntax

cm *expression*

co *expression*

cs *expression*

expression An expression (of the appropriate type) to be converted.

Examples

```
>cs 0
stsOK
>cs 081000002
stsBadObject
```

ctx

Sets DB's context. Sets the current task and sets the lexical scope. When no parameters are specified, **ctx** displays the current context.

Syntax

[*taskID*] **ctx** [*scopeSpec*]

taskID A task ID as described in Section 7.2.7. If no *taskSet* is specified, the current task is operated upon. The set of current task IDs is available via the **tl** command.

scopeSpec A scopeSpec as defined in Section 7.2.1. Defaults to the top of the task's call stack.

Examples

All of the following examples assume that task 0508 has the following stack trace:

```
"go-calculator-v1(0)"[1] 0508>st
%1 >
CalcEngineEvalBinary(
    opnd1: 99.000000, op: 5, opnd2: 99.000000, pResult: 0432bbc)
%2
CalcEngineEnterOperator(
    msg: 25170108, self: 01ad203fb, pArgs: 043444c, xxx: 0432be8,
    pData: 043ff0608)
%3  ObjectCall [ msgCalcEngineEnterOperator [dyn 1ad203fb] 043444c 0 ]
%4
CalcEngineProcessKey(
    msg: 8392892, self: 01ad203fb, pArgs: 043444c, xxx: 0432c48,
    pData: 043ff0608)
%5  ObjectCall [ msgCalcEngineProcessKey [dyn 1ad203fb] 043444c 043a4c8d5 ]
%6
CalcAppButtonNotify(
    msg: 117440616, self: 01ac503e5, pArgs: 05, xxx: 0432ca0, pData: 043fefeee)
%7  ObjectCall [ msgButtonNotify [dyn 1ac503e5] 05 01ada0402 ]
%8  ButtonNotifyClient [ 01ada0402 043ff0bec 0432d98 0409bc610 ]
...
```

Sets DB's context to a specific task. Note that DB's prompt changes.

```
>0508 ctx
"go-calculator-v1(0)"[1] 0508>
```

Displays DB's current context.

```
"go-calculator-v1(0)"[1] 0508>ctx
%1   "calc.dll"."calceng.obj".CalcEngineEvalBinary."calceng.c".@80
```

Causes DB to move out to the scope of CalcEngineEnterOperator.

```
"go-calculator-v1(0)"[1] 0508>ctx outto.CalcEngineEnterOperator
"go-calculator-v1(0)"[1] 0508>ctx
%2   "calc.dll"."calceng.obj".CalcEngineEnterOperator."calceng.c".@417
```

The following example move's DB context to the top of the stack and then out two stack frames. The context is then moved in one stack frame.

```
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>ctx out.out
"go-calculator-v1(0)"[1] 0508>ctx
%3   go-kernel3-V0(1).01: 06bea
"go-calculator-v1(0)"[1] 0508>ctx in
"go-calculator-v1(0)"[1] 0508>ctx
%2   "calc.dll"."calceng.obj".CalcEngineEnterOperator."calceng.c".@417
```

The following move the context out the stack to the frame corresponding to **CalcAppButtonNotify**.

```
"go-calculator-v1(0)"[1] 0508>ctx outto.CalcAppButtonNotify
"go-calculator-v1(0)"[1] 0508>ctx
%6   "calc.exe"."calcapp.obj".CalcAppButtonNotify."calcapp.c".@324
```

Causes DB to switch to the fourth stack frame.

```
"go-calculator-v1(0)"[1] 0548>ctx %4
"go-calculator-v1(0)"[1] 0548>ctx
%4   "calc.dll"."calceng.obj".CalcEngineProcessKey."calceng.c".@482
```

Causes DB to switch to the task 0508, and position to the top of that task's call stack.

```
> 0508ctx top
```

This variation of the context command makes the set of ids defined in the "calculator" module be the set of ids currently visible. Notice that the prompt does *not* change. (See the Datasheet for the **ids** command for more information.)

```
>ctx "calculator"
>ids
{ (tags) APP_NEW APP_NEW_ONLY APP_OPEN CALC_APP_DATA CALC_ENGINE_TOKEN
CLS_SYM_MSG CLS_SYM_OBJ CLS_SYM_STS FS_LOCATOR OBJ_RESTORE OBJ_SAVE
OBJECT_NEW (vars) calcMsgSymbols calcObjSymbols calcStsSymbols
clsCalcAppTable (functions) CalcAppAppInit CalcAppButtonNotify CalcAppClose
CalcAppCreateButtons CalcAppCreateCalcWindow CalcAppCreateDisplayWindow
CalcAppDisplayString CalcAppFree CalcAppInit CalcAppLayoutButtons CalcAppOpen
CalcAppRestore CalcAppSave CalcSymbolsInit ClsCalcAppInit main}
```

Remarks

When changing tasks, the **ctx** command changes DB's prompt so that it reflects the new task.

d, db, dw, dd

Display memory contents

Syntax

d [**b** | **w** | **d**] [*dataAddress*] [*length*]

d Display memory contents. Without one of the following options, defaults to the display format of the previous **d**, **db**, **dw**, or **dd** command.

db Display memory contents as bytes.

dw Display memory contents as 2 byte words, which are displayed as four hexadecimal digits. DB assumes the Intel reversed-word ordering—the most significant byte of the word comes second in address space.

dd Display memory contents as 4 byte double-words, which are displayed as eight hexidecimal digits. DB assumes the Intel reversed-word ordering for double-words and within 16-bit words—the most significant word in the double-word comes second in address space.

dataAddress Start address for the display, as described in Section 7.2.4. If omitted, DB starts from where it left off in the previous **d**, **db**, **dw**, or **dd** command.

length Number of bytes to display.

Examples

```
"go-calculator-v1(0)"[1] 0508>db pArgs
434440                         -          05 00 31 00              ..1.
434450  30 30 30 30 30 30 30 30-30 AC 07 00 D8 05 04 A8  000000000,..X..(
434460  AC 07 00 CD 05 04 48 5F-07 00 C3 05 00 00 00 00  ,..M..H_..C.....
434470  00 C0 58 40 FB 03 D2 1A-FC 03 D4 1A FD 03 D5 1A  .@X@{.R.|.T.}.U.
434480  FE 03 D6 1A AE AE 08 05-72 FA 02 E0              ~.V.....rz.'
"go-calculator-v1(0)"[1] 0508>dw pArgs
434440                                0005 0031
434450  3030 3030 3030 3030 AC30 0007 05D8 A804
434460  07AC CD00 0405 5F48 0007 05C3 0000 0000
434470  C000 4058 03FB 1AD2 03FC 1AD4 03FD 1AD5
434480  03FE 1AD6 AEAE 0508 FA72 E002
"go-calculator-v1(0)"[1] 0508>dd pArgs
434440                             00310005
434450  30303030 30303030 0007AC30 A80405D8
434460  CD0007AC 5F480405 05C30007 00000000
434470  4058C000 1AD203FB 1AD403FC 1AD503FD
434480  1AD603FE 0508AEAE E002FA72
"go-calculator-v1(0)"[1] 0508>d pArgs
434440                             00310005
434450  30303030 30303030 0007AC30 A80405D8
434460  CD0007AC 5F480405 05C30007 00000000
434470  4058C000 1AD203FB 1AD403FC 1AD503FD
434480  1AD603FE 0508AEAE E002FA72
```

dp

Displays profile data.

Remarks

See Chapter 8, which discusses profiling and the **dp** command in detail.

files

Display the files associated with a scope.

Syntax

files [*scopeSpec*]

scopeSpec A scopeSpec as defined in Section 7.2.1. If not specified, **files** displays all of the known files.

Examples

```
>srcdir "calc.dll" \\c\penpoint\sdk\sample\calc
>srcdir "calc.exe" \\c\penpoint\sdk\sample\calc
>files
"calc.dll"
"calc.dll"."cengmeth.obj" (obj)
"calc.dll"."calceng.obj" (obj)
"calc.dll"."calceng.obj"."calceng.c" (src) in \\c\penpoint\sdk\sample\calc
"calc.dll"."apprefs.obj" (obj)
"calc.exe"
"calc.exe"."cappmeth.obj" (obj)
"calc.exe"."calcapp.obj" (obj)
"calc.exe"."calcapp.obj"."calcapp.c" (src) in \\c\penpoint\sdk\sample\calc
"calc.exe"."s_calc.obj" (obj)
"calc.exe"."s_calc.obj"."s_calc.c" (src) in \\c\penpoint\sdk\sample\calc
"calc.exe"."appstart.obj" (obj)
```

Remarks

This command displays a subset of the information displayed by the **ids** command.

fl

Lists values of debugging flags.

Syntax

fl [*flagSet*]

flagSet Either a single printing character or two hex digits. If missing, **fl** displays the value of all debugging flags.

Examples

```
>fl
42 B 00000769    44 D 00008000    47 G 00000040    5a Z 08000000
```

Remarks

Only those flags which contain at least one non-zero value are displayed.

See the Datasheet for **fs** for a discussion of setting debugging flags.

fns

Display the functions known in a scope.

Syntax

fns [*scopeSpec*]

scopeSpec A scopeSpec as defined in Section 7.2.1. Defaults to showing the functions in the current scope and each of its parent scopes.

Examples

```
"go-calculator-v1(0)"[1] 0508>fns
{}
{}
{}
{ (functions) CalcEngineAppendCharToToken CalcEngineDump
CalcEngineEnterNumber CalcEngineEnterOperator CalcEngineEvalBinary
CalcEngineEvalUnary CalcEngineFree CalcEngineInit CalcEngineProcessKey
CalcEngineRestore CalcEngineSave CalcEngineSaveValue CalcEngineUpdateToken
DllMain}
```

```
"go-calculator-v1(0)"[1] 0508>fns "calc_eng"
{ (functions) CalcEngineAppendCharToToken CalcEngineDump
CalcEngineEnterNumber CalcEngineEnterOperator CalcEngineEvalBinary
CalcEngineEvalUnary CalcEngineFree CalcEngineInit CalcEngineProcessKey
CalcEngineRestore CalcEngineSave CalcEngineSaveValue CalcEngineUpdateToken
DllMain}
```

Remarks

This command displays a subset of the information displayed by the **ids** command.

fs

Sets values of debugging flags.

Syntax

fs *flagSet* [[+ |] *flagValue*]

flagSet Either a single printing character or two hex digits.

flagValue A hex value. If not present, **fs** displays the current value of *flagSet*.

\+ If present, turn the bits in *flagValue* on.

– If present, turn the bits in *flagValue* off.

Examples

```
>fl G
040
>fs G 44
>fl G
044
>fl G + 800
>fl G
0844
>fs G 40
>fl G
040
```

Remarks

See the Datasheet for **fl** for a discussion of listing the values of debugging flags.

g

Resume PenPoint execution.

Syntax

[*taskSet*] **g** [*codeAddress*]

taskSet A task set as defined in Section 7.2.7. Defaults to the current task.

codeAddress If specified, a temporary breakpoint is set at the code address. Execution continues until this or any other breakpoint is encountered. The address is specified as described in Section 7.2.3.

Examples

Starts execution of the current task after setting a breakpoint at the entry point for **CalcEvalBinary**.

```
>g CalcEngineEvalBinary
```

Remarks

Execution continues until PenPoint detects a ***fault condition***, or DB detects a ***breakpoint***, or a task in ***taskSet*** terminates.

Only tasks with non-zero freeze counts will run.

h

Display help on DB commands and topics.

Syntax

h *topic*

Examples

Type the following to get a list of available topics:

```
>h
```

Type the following to get a list of DB commands:

```
>h commands
```

id

Display type and declaration information for an identifier.

Syntax

id { *identifier* | *address* }

identifier A function or variable identifier from your program.

address DB will try to find an identifier near this address.

Examples

```
"go-calculator-v1(0)"[1] 0508>id pInst
program local pInst: ptr to struct CALC_ENGINE_DATA @ [bp-16]

"go-calculator-v1(0)"[1] 0508>id msg
program stack parameter msg: long @ [bp+8]

"go-calculator-v1(0)"[1] 0508>id pData
program stack parameter pData: ptr to ptr to struct CALC_ENGINE_DATA @ [bp+24]

"go-calculator-v1(0)"[1] 0508>id CalcEngineEnterOperator
program global CalcEngineEnterOperator:
function(
    msg: long, self: ptr to void, pArgs: ptr to struct CALC_ENGINE_TOKEN, ctx:
    ptr to void, pData: ptr to ptr to struct CALC_ENGINE_DATA, ...)
returning long @ go-calc_eng-v1(0) 1.926
```

ids

Display the identifiers known in a scope.

Syntax

ids [*scopeSpec*]

scopeSpec A scopeSpec as defined in Section 7.2.1. Defaults to the current scope and all of its parent scopes.

Examples

```
"go-calculator-v1(0)"[1] 0508>ids
{ (vars) pInst result s}
{ (params) msg self pArgs ctx pData}
{}
{ (tags) CALC_ENGINE_DATA CALC_ENGINE_NEW CALC_ENGINE_TOKEN OBJ_RESTORE
OBJ_SAVE OBJECT_NEW (vars) clsCalcEngineTable (functions) _8087
CalcEngineAppendCharToToken CalcEngineDump CalcEngineEnterNumber
CalcEngineEnterOperator CalcEngineEvalBinary CalcEngineEvalUnary
CalcEngineFree CalcEngineInit CalcEngineProcessKey CalcEngineRestore
CalcEngineSave CalcEngineSaveValue CalcEngineUpdateToken DllMain}
```

```
"go-calculator-v1(0)"[1] 0548>ids %3
{ (vars) s}
{ (params) msg self pArgs ctx pData}
```

Remarks

The volume of information displayed by the **ids** command can be overwhelming. You might want to use the **fns** or **vars** commands to get a subset of the information displayed by **ids.**

k

Display the call stack.

The **k** command is a synonym for the **st** command; see the **st** command's Datasheet.

log

Starts and stops logging PenPoint debugging input and output.

Syntax

log [[+] *filename*]

filename The full pathname for the log file.

\+ Optionally specifies that the log be appended to an existing file.

With no parameters, **log** turns off logging.

mi

Display memory information.

Syntax

mi

Examples

```
"go-calculator-v1(0)"[1] 0508>mi
allocated:  217088 task local  233472 task total  6328320 sys total
regions:         4 task local       5 task total      939 sys total (0 shr)
28672 task resident 184320 swapped
8257536 memory 7925760 allocatable 1372160 nonswappable 1597440 free
5242880 swap 2834432 free (page size = 4096)
```

mini

Enters PenPoint's mini-debugger.

Syntax **mini**

Remarks To return to DB, type g to the mini-debugger.

od

Send msgDump to an object.

Syntax **od** *object* [*pArgs*]

object Expression that evaluates to an object.

pArgs pArgs sent with the message. Defaults to –1.

Examples **>od theRootWindow**

```
msgDump(clsObject): object=theRootWindow {cls=[dyn 01c80094]}, caps=0x8618
  parent=window has no parent, first child=[dyn 16f6034a](34)
  bounds (x,y,w,h)=0,0,640,480
  updateNesting=0, synchVer=6, tag=0
  flags.style:
wsClipChildren wsClipSiblings
wsVisible wsPaintable
Dirty region is:
411a12d8: yRange = 1    1, 1 x-range slots max, 0 slots allocated
```

>od theProcess

```
msgDump(clsObject): object=theProcess {cls=clsProcess}, caps=0x8018
msgDump(clsProcess): local well-known theProcess, process=0x0128
```

on

Creates a "handler" for specified events.

Most variations of the **on** command are needed only by advanced or specialized DB users. See the Advanced DB Techniques chapter for more information on the **on** command.

on access, on store

Causes execution to halt when memory is read and/or written.

Syntax [*taskSet*] **on store** *lvalue* [*eventAction*]

taskSet A task set as defined in Section 7.2.7. Defaults to all tasks.

lvalue Typically a variable name, although it can also be any expression referencing a region of storage. (Consult any high quality C language reference manual for a more complete description.)

eventAction C code to be executed when the breakpoint is hit.

Examples

```
"go-calculator-v1(0)"[1] 0508>on store (*pData)->numberEntered
"go-calculator-v1(0)"[1] 0508>on access (*pData)->numberEntered
"go-calculator-v1(0)"[1] 0508>on access *(short *)043477c
```

Remarks Only a limited amount of memory may be watched at any given time.

Execution halts *after* the instruction that reads and/or writes memory is executed. This is in contrast to breakpoints (created with the **bp** command), which halt execution *before* the breakpointed instruction.

There are other variations of the **on** command. See the Datasheet and the appropriate sections in the Advanced DB Techniques chapter.

p, P

Single-steps execution, *passing over* called routines.

p single-steps through *source* statements. **P** single-steps through *assembly* instructions.

Syntax

[*taskSet*] **p** [*count*]
[*taskSet*] **P** [*count*]

taskSet A task set as defined in Section 7.2.7. Defaults to the current task.

count Number of trace steps.

Examples

Executes one source statement.

```
"go-calculator-v1(0)"[1] 0508>p
   78>>     switch (op) {
```

Executes 10 source statements.

```
"go-calculator-v1(0)"[1] 0508>p 10
   80>>             if (opnd2 == 0.0) {
   83>>                 *pResult = opnd1 / opnd2;
   84>>             }
  104>>     return s;
  106>> } /* CalcEngineEvalBinary */
  418>>             pInst->pendingOp = nop;
  419>>             CalcEngineSaveValue(result, pArgs, pInst);
  421>>         pInst->pendingOp = pArgs->key;
  422>>         pInst->numberEntered = false;
  426>>     return stsOK;
```

Executes one assembly instruction.

```
"go-calculator-v1(0)"[1] 0508>P
43A1C983  89 EC             MOV   ESP, EBP
```

Executes 10 assembly instructions.

```
"go-calculator-v1(0)"[1] 0508>P 10
43A1C985  5D                POP   EBP
43A1C986  C3                RET
E002D7FF  83 C4 14          ADD   ESP, 20
E002D802  8D 65 F4          LEA   ESP, [EBP-12]
E002D805  5F                POP   EDI
E002D806  5E                POP   ESI
E002D807  5B                POP   EBX
E002D808  5D                POP   EBP
E002D809  C2 0C 00          RET   12
43A1CABB  89 45 F4          MOV   DWORD PTR [EBP-12{s}], EAX
```

Remarks

p and **P** differ from **t** and **T** in that **p**/**P** do not step into called routines while **t**/**T** step *into* routines.

Other un-frozen tasks may execute before the next source statement is reached.

If the **p** command is given when DB does not have source code information, it behaves just like the **P** command.

profile

Create a profile breakpoint

Remarks

See Chapter 8, which discusses profiling and the **profile** command in detail.

q

Quit DB.

Syntax

q

r

Display the registers.

Syntax

[*taskID*] **r**

taskID A task ID as defined in Section 7.2.7. Defaults to the current task.

Examples

```
"go-calculator-v1(0)"[1] 0508>r
eax 00000000 ebx 008010BC ecx 41D00031 edx 00430005 esi 43FE9538 edi 418F34F8
esp 00432C20 ebp 00432C2C eip 43A1CABB (user) ---0--IT---P-
```

srcdir

Tells DB where to find source files for a module.

Syntax

srcdir [*scopeSpec* [*dirName*]]

scopeSpec The identifier by which the module is known to PenPoint at load time; described in Section 7.2.1. Generally, you specify a .EXE, .DLL, .C, or .OBJ file.

dirName The path to the directory containing the source for the module.

Examples

Specifies the source directory for the "calc_eng" module.

```
>srcdir "calc_eng" \\c\penpoint\sdk\sample\calc
```

Without a dirName, **srcdir** gives the current source directory, if any, for a module.

```
>srcdir "calc_eng"
"calc.dll"  \\c\penpoint\sdk\sample\calc
```

Remarks

If you do not specify any arguments, **srcdir** shows all source directory settings.

Specifying "*" for the scopeSpec sets the default source directory for all modules not otherwise set with a **srcdir** command.

If DB finds no other **srcdir** set for a particular .C file, it checks the parent .OBJ file. If DB finds no **srcdir** set for a particular .OBJ file, it checks its parent .EXE file. If DB finds no **srcdir** set for a particular .EXE file, it use the default **srcdir**.

Alternatively, you can set the DBSrc environment variable in \PENPOINT\BOOT\ENVIRON.INI. DB will search these directories *after* searching any directories specified in **srcdir** commands.

```
DBSrc=\\c\dir1;\\c\dir2;\\c\dir3
```

st

Display the call stack.

Syntax

[*taskID*] **st** { [**+** | **–**] [**c** | **l** | **p**] }
[*scopeSpec*] [*count*]

taskID A task set as defined in Section 7.2.7. Defaults to the current task.

+/– c Controls display of context lines. These are the identifiers that are listed with varying indent to show their nesting levels.

+/– l Controls display of block-local variables.

+/– p Controls display of function parameters.

scopeSpec A scopeSpec as defined in Section 7.2.1.

count Number of stack frames to display.

The default flags are *+p –l –c*.

Examples

```
"go-calculator-v1(0)"[1] 0508>st
%1 >
CalcEngineEvalBinary(opnd1: 1.200000, op: 5, opnd2: 9.000000, pResult: 0432bbc)
%2
CalcEngineEnterOperator(
    msg: 25170108, self: 01ad103fb, pArgs: 043444c, ctx: 0432be8,
    pData: 043fe9538)
%3  ObjectCall [ msgCalcEngineEnterOperator [dyn 1ad103fb] 043444c 0 ]
%4
CalcEngineProcessKey(
    msg: 8392892, self: 01ad103fb, pArgs: 043444c, ctx: 0432c48,
    pData: 043fe9538)
%5  ObjectCall [ msgCalcEngineProcessKey [dyn 1ad103fb] 043444c 043a478d5 ]
%6
CalcAppButtonNotify(
    msg: 117440616, self: 01ac403e5, pArgs: 05, ctx: 0432ca0, pData: 043fe8e96)
%7  ObjectCall [ msgButtonNotify [dyn 1ac403e5] 05 01ad90402 ]
%8  ButtonNotifyClient [ 01ad90402 043fe9b1c 0432d98 0409bc610 ]
...
"go-calculator-v1(0)"[1] 0508>st +c p l
%1 > "calc.dll"."calceng.obj".CalcEngineEvalBinary."calceng.c".@76
%2  "calc.dll"."calceng.obj".CalcEngineEnterOperator."calceng.c".@413
%3  0e002d7ff
%4  "calc.dll"."calceng.obj".CalcEngineProcessKey."calceng.c".@478
%5  0e002d7ff
%6  "calc.exe"."calcapp.obj".CalcAppButtonNotify."calcapp.c".@320
%7  0e002d7ff
%8  go-tk-v1(0).01:01371b
...
"go-calculator-v1(0)"[1] 0508>st +l
%1 > (block) {s: 4409164;}
CalcEngineEvalBinary(opnd1: 1.200000, op: 5, opnd2: 9.000000, pResult: 0432bbc)
%2  (block) {pInst: 043474c; result: 2.132846e-307; s: 1140757816;}
CalcEngineEnterOperator(
    msg: 25170108, self: 01ad103fb, pArgs: 043444c, ctx: 0432be8,
    pData: 043fe9538)
%3  ObjectCall [ msgCalcEngineEnterOperator [dyn 1ad103fb] 043444c 0 ]
%4  (block) {s: 0;}
CalcEngineProcessKey(
```

```
    msg: 8392892, self: 01ad103fb, pArgs: 043444c, ctx: 0432c48,
    pData: 043fe9538)
%5   ObjectCall [ msgCalcEngineProcessKey [dyn 1ad103fb] 043444c 043a478d5 ]
%6   (block) {s: 1134852309;}
CalcAppButtonNotify(
    msg: 117440616, self: 01ac403e5, pArgs: 05, ctx: 0432ca0, pData: 043fe8e96)
%7   ObjectCall [ msgButtonNotify [dyn 1ac403e5] 05 01ad90402 ]
%8   ButtonNotifyClient [ 01ad90402 043fe9b1c 0432d98 0409bc610 ]
...
"go-calculator-v1(0)"[1] 0508>st p
%1 > "calc.dll"."calceng.obj".CalcEngineEvalBinary
%2   "calc.dll"."calceng.obj".CalcEngineEnterOperator
%3   0e002d7ff
%4   "calc.dll"."calceng.obj".CalcEngineProcessKey
%5   0e002d7ff
%6   "calc.exe"."calcapp.obj".CalcAppButtonNotify
%7   0e002d7ff
%8   go-tk-v1(0).01:01371b
...
```

Remarks

With no parameters, **st** continues from where the previous **st** command left off.

The **st** command lists the execution contexts pushed onto the call stack for the current task up to the last instruction executed.

DB records function calls and moves into nested blocks on the call stack. Within each context, the variables local to it and any parameters defined at its entry are listed with their current values.

sym

Loads symbolic debugging information into memory.

Syntax

sym [*scopeSpec* [(*objFileList*)]]

scopeSpec A scope spec that identifies a load module. Described in section 7.2.1

objFileList This is an optional list of comma-separated object file names (without the trailing ".obj"). If present, DB reads the detailed symbol information for only those object files, thereby saving substantial amounts of memory. The **objFileList** must be separated from the **scopeSpec** by a space. If *objFileList* consists of just "()" (the parentheses with no files between them), the command loads only the public identifiers and line numbers of the module.

Examples

The following three commands load all of the symbols in the module:

>sym "foo"
>sym "go-foo-v1"

The following command will load full symbolic debugging information for obj1.obj and obj2.obj, but only line numbers and global identifiers for the rest of the module's object files. (The space before the "(" is mandatory.)

>sym "go-foo-v1" (obj1,obj2)

Remarks

With no arguments, **sym** displays all programs and their associated symbol file, if any.

t, T

Single-steps execution, stepping into called routines.

t single-steps through *source* statements. **T** single-steps through *assembly* instructions.

Syntax

[*taskSet*] **t** [*count*]
[*taskSet*] **T** [*count*]

taskSet A task set as defined in Section 7.2.7. Defaults to the current task.

count Number of trace steps.

Examples

Executes one source statement.

```
"go-calculator-v1(0)"[1] 0508>t
   78>>     switch (op) {
```

Executes 10 source statements.

```
"go-calculator-v1(0)"[1] 0508>t 10
   80>>             if (opnd2 == 0.0) {
   83>>                 *pResult = opnd1 / opnd2;
   84>>             }
  104>>     return s;
  106>> } /* CalcEngineEvalBinary */
  418>>             pInst->pendingOp = nop;
  419>>             CalcEngineSaveValue(result, pArgs, pInst);
  161>> CalcEngineSaveValue (
  166>>     if (pInst->pendingOp == nop) {
  167>>         pInst->xValue = value;
```

Executes one assembly instruction.

```
"go-calculator-v1(0)"[1] 0508>T
43A1C3A7  8B 55 14          MOV   EDX, DWORD PTR [EBP+20{pInst}]
```

Executes 10 assembly instructions.

```
"go-calculator-v1(0)"[1] 0508>T 10
43A1C3AA  89 02             MOV   DWORD PTR [EDX], EAX
43A1C3AC  8B 45 0C          MOV   EAX, DWORD PTR [EBP+12{value+4}]
43A1C3AF  8B 55 14          MOV   EDX, DWORD PTR [EBP+20{pInst}]
43A1C3B2  89 42 04          MOV   DWORD PTR [EDX+4], EAX
43A1C3B5  8B 45 08          MOV   EAX, DWORD PTR [EBP+8{value}]
43A1C3B8  8B 55 14          MOV   EDX, DWORD PTR [EBP+20{pInst}]
43A1C3BB  89 42 08          MOV   DWORD PTR [EDX+8], EAX
43A1C3BE  8B 45 0C          MOV   EAX, DWORD PTR [EBP+12{value+4}]
43A1C3C1  8B 55 14          MOV   EDX, DWORD PTR [EBP+20{pInst}]
43A1C3C4  89 42 0C          MOV   DWORD PTR [EDX+12], EAX
```

Remarks

t and **T** differ from **p** and **P** in that **t**/**T** step *into* routines while **p**/**P** do not step into called routines.

Other un-frozen tasks may execute before the next source statement is reached.

If the t command is given when DB does not have source code information, it behaves just like the T command.

ti

Display task information.

Syntax

[*taskSet*] **ti**

taskSet Set of tasks to show information for as described in Section 7.2.7. Defaults to the current task.

Examples

```
"go-calculator-v1(0)"[1] 0508>ti
 00508 P "go-calculator-v1(0)"[1]                      1.439 Step
```

Remarks

The **ti** command is just like the **tl** command, except that the default is different. See the Datasheet of the **tl** command for more information about **ti**'s output.

tl

Display PenPoint's task list.

Syntax

[*taskSet*] **tl**

taskSet An optional task set as described in Section 7.2.7. DB displays the status of each task in the set. Defaults to all tasks.

Examples

```
>tl
 00508 P "go-calculator-v1(0)"[1]                      1.439 Step
 004e8 P "go-nbtoc-v1"[1]                               .907 Msg
 004d8 P "go-nbapp-v1"[1]                               .886 Msg
 004c8 P "go-dtapp-v1"[1]                              1.134 Msg
 004b8 P "go-calculator-v1(0)"[0]                       .154 Msg
 004a8 P "go-calc_eng-v1(0)"[0]                          .007 Susp Msg
 ...
 00288 P "go-minitext-v1"[0]                            .372 Msg
 00268 P "go-notepaperapp-v1"[0]                        .199 Susp Msg
 00258 P "go-notepaper-v1(0)"[0]                        .041 Susp Msg
 00238 P "go-nbapp-v1"[0]                               .126 Susp Msg
 00218 P "go-nbtoc-v1"[0]                               .127 Susp Msg
 001f8 P "go-sectapp-v1"[0]                             .170 Susp Msg
```

Remarks

The first column of output displays PenPoint's task ID. The second column contains either a P for Process or S for Subtask. The third column displays DB's name for the task. The fourth column contains the accumulated runtime of the task. The fifth column contains a mnemonic indicating the state of the task. Common mnemonics and their meanings are:

Msg Task is waiting for a message.
Sem Task is waiting for a message.
SSm Task is waiting for a system semaphore.
FSm Task is waiting for a fast semaphore.
Tmr Task is waiting for a timer.
BP Task is waiting for a breakpoint.
Step Task is executing a single step.
Rdy Task is ready to run.

The fifth column may also include the **Susp** keyword; if so, the task is suspended.

type

Display type that a C expression evaluates to.

Syntax

type *C-expression*

C-expression Any C expression using variables that are defined in the current context, or any identifier known to the current context or fully referenced.

Examples

```
>type 12
long

>type "hello"
array [6] of char

"go-calculator-v1(0)"[1] 0508>type self
ptr to void

"go-calculator-v1(0)"[1] 0508>type pData
ptr to ptr to struct CALC_ENGINE_DATA

"go-calculator-v1(0)"[1] 0508>type *pData
ptr to struct CALC_ENGINE_DATA

"go-calculator-v1(0)"[1] 0508>type **pData
struct CALC_ENGINE_DATA {
    xValue: double; yValue: double; pendingOp: short; keysSeen:
    array [30] of unsigned char; numberEntered: short; calcError: short;}

>type CalcEngineEvalBinary
function(opnd1: double, op: short, opnd2: double, pResult: ptr to double, ...)
returning long
```

u

View assembly code.

Syntax

u [*codeAddress* | *scopeSpec*] [*lineCount*]

codeAddress A code address as defined in Section 7.2.3. Defaults to where the previous **u** command left off.

scopeSpec A scopeSpec as defined in Section 7.2.1.

lineCount A lineCount as defined in Section 7.2.2.

Examples

The following shows the assembly code that makes up the first part of the routine CalcEngineEvalBinary.

```
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>u
43A1C19A  C7 45 F8 00 00 00 00
                            MOV   DWORD PTR [EBP-8{s}], 0
43A1C1A1  66 8B 45 10       MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02       SUB   AX, 2
43A1C1A9  66 3D 03 00       CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00 JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0          MOVZX EAX, AX
43A1C1B6  C1 E0 02          SHL   EAX, 2
43A1C1B9  2E FF A0 70 C1 A1 43
                            JMP   DWORD PTR CS:[EAX-16016]
43A1C1C0  83 7D 18 00       CMP   DWORD PTR [EBP+24{opnd2+4}], 0
43A1C1C4  75 0C             JNZ   043A1C1D2{CalcEngineEvalBinary.@83}
```

You can also specify a **codeAddress** for the **u** command.

```
"go-calculator-v1(0)"[1] 0508>u CalcEngineEvalBinary 1 10
43A1C195  55                PUSH  EBP
43A1C196  89 E5             MOV   EBP, ESP
43A1C198  53                PUSH  EBX
43A1C199  50                PUSH  EAX
43A1C19A  C7 45 F8 00 00 00 00
                            MOV   DWORD PTR [EBP-8{s}], 0
43A1C1A1  66 8B 45 10       MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02       SUB   AX, 2
43A1C1A9  66 3D 03 00       CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00 JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0          MOVZX EAX, AX
```

Remarks

You should not specify a data address; you will get spurious results

Where possible, the u command inserts an identifier after addresses that it can interpret symbolically.

Repeating the **u** command displays the next several lines of source code. In this way you can page down through a source file.

---

uv

View assembly code with interspersed source code.

Syntax

The syntax of the **uv** command is identical to that of the **u** command.

Examples

The following shows the assembly code that makes up the first part of the routine **CalcEngineEvalBinary**, with source code statements interspersed.

```
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>uv
   76       STATUS              s = stsOK;
   77
43A1C19A  C7 45 F8 00 00 00 00
                            MOV   DWORD PTR [EBP-8{s}], 0
   78       switch (op) {
   79           case divide:
43A1C1A1  66 8B 45 10       MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02       SUB   AX, 2
43A1C1A9  66 3D 03 00       CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00 JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0          MOVZX EAX, AX
43A1C1B6  C1 E0 02          SHL   EAX, 2
43A1C1B9  2E FF A0 70 C1 A1 43
                            JMP   DWORD PTR CS:[EAX-16016]
   80               if (opnd2 == 0.0) {
43A1C1C0  83 7D 18 00       CMP   DWORD PTR [EBP+24{opnd2+4}], 0
43A1C1C4  75 0C             JNZ   043A1C1D2{CalcEngineEvalBinary.@83}
```

You can also specify a codeAddress for the **uv** command.

```
"go-calculator-v1(0)"[1] 0508>uv CalcEngineEvalBinary 1 10
   70   CalcEngineEvalBinary (
   71       double              opnd1,
   72       CALC_ENGINE_KEY     op,
   73       double              opnd2,
   74       double *            pResult)
   75   {
43A1C195  55                PUSH  EBP
43A1C196  89 E5             MOV   EBP, ESP
43A1C198  53                PUSH  EBX
43A1C199  50                PUSH  EAX
   76       STATUS              s = stsOK;
```

```
   77
43A1C19A  C7 45 F8 00 00 00 00
                                 MOV   DWORD PTR [EBP-8{s}], 0
   78        switch (op) {
   79            case divide:
43A1C1A1  66 8B 45 10            MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02            SUB   AX, 2
43A1C1A9  66 3D 03 00            CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00      JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0               MOVZX EAX, AX
```

Remarks

The mapping between assembly code and source code is, by its nature, imprecise. It is particularly imprecise when the compiler's optimization is turned on.

The **uv** command differs from the **vu** command in that the **uv** command always list the assembly code properly and intersperses the source code as best it can given the imprecise nature of the task.

See Remarks in the **u** Datasheet.

---

v

View source code.

Syntax

v [*codeAddress* | *scopeSpec*] [*lineCount*]

codeAddress A code address as defined in Section 7.2.3. Defaults to where the previous v command left off.

scopeSpec A scopeSpec as defined in Section 7.2.1.

lineCount A lineCount as defined in Section 7.2.2.

Examples

```
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>v
   73        double              opnd2,
   74        double *            pResult)
   75    {
   76>>      STATUS              s = stsOK;
   77
   78        switch (op) {
   79            case divide:
   80                if (opnd2 == 0.0) {

"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>v 1 10
   76>>      STATUS              s = stsOK;
   77
   78        switch (op) {
   79            case divide:
   80                if (opnd2 == 0.0) {
   81                    s = stsCalcEngineComputeError;
   82                } else {
   83                    *pResult = opnd1 / opnd2;
   84                }
   85                break;
```

Remarks

Repeating the v command displays the next several lines of source code. In this way you can page down through a source file.

vars

Display variables known in a scope.

Syntax

vars [*scopeSpec*]

scopeSpec A *scopeSpec* as defined in Section 7.2.1. Defaults to showing the variables in the current scope and each of its parent scopes. If you specify "top", vars displays the variables declared interactively with the debugger.

Examples

```
"go-calculator-v1(0)"[1] 0508>vars
{ (vars) s}
{ (params) opnd1 op opnd2 pResult}
{}
{ (vars) clsCalcEngineTable}
"go-calculator-v1(0)"[1] 0508>vars %6
{ (vars) s}
{ (params) msg self pArgs ctx pData}
```

Remarks

This command displays a subset of the information displayed by the **ids** command.

ver

Display DB's version.

Syntax

ver

vu

View source code with interspersed assembly language statements.

Syntax

The syntax of the **vu** command is identical to that of the **v** command.

Examples

```
"go-calculator-v1(0)"[1] 0508>ctx top
"go-calculator-v1(0)"[1] 0508>vu 1 4
   76>>     STATUS                s = stsOK;
43A1C19A  C7 45 F8 00 00 00 00
                              MOV   DWORD PTR [EBP-8{s}], 0
   77
43A1C19A  C7 45 F8 00 00 00 00
                              MOV   DWORD PTR [EBP-8{s}], 0
   78       switch (op) {
43A1C1A1  66 8B 45 10         MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02         SUB   AX, 2
43A1C1A9  66 3D 03 00         CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00   JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0            MOVZX EAX, AX
43A1C1B6  C1 E0 02            SHL   EAX, 2
43A1C1B9  2E FF A0 70 C1 A1 43
                              JMP   DWORD PTR CS:[EAX-16016]
   79           case divide:
43A1C1A1  66 8B 45 10         MOV   AX, WORD PTR [EBP+16{op}]
43A1C1A5  66 83 E8 02         SUB   AX, 2
43A1C1A9  66 3D 03 00         CMP   AX, 3
43A1C1AD  0F 87 97 00 00 00   JA    043A1C24A{CalcEngineEvalBinary.@100}
43A1C1B3  0F B7 C0            MOVZX EAX, AX
43A1C1B6  C1 E0 02            SHL   EAX, 2
43A1C1B9  2E FF A0 70 C1 A1 43
                              JMP   DWORD PTR CS:[EAX-16016]
```

Remarks

The mapping between assembly code and source code is, by its nature, imprecise. It is particularly imprecise when the compiler's optimization is turned on.

The **vu** command differs from the **uv** command in that the **vu** command always list the source code properly and intersperses the assembly code as best it can given the imprecise nature of the task.

See Remarks in the v Datasheet.

---

zp

Clears profile data.

Remarks

See Chapter 8, which discusses profiling and the **zp** command in detail.

Chapter 8 / Profiling with DB

This chapter contains information about using DB to profile program execution. This can help you find performance bottlenecks in your program.

Profile Breakpoints 8.1

Profile information is maintained in *profile breakpoints*. Many breakpoint commands operate on profile breakpoints as well as normal breakpoints; consult the appropriate Datasheet for complete information.

- bl
- bc
- bd
- be

Two Types of Profiles 8.2

DB can collect two types of profiles:

Code Profiles This type of profile collects execution information based on routines or line numbers.

Object Profile This type of profile collects execution information based on objects and messages.

With both types, DB collects information into profiling *buckets*. You can control the number (or granularity) of the buckets and what gets recorded in each bucket.

Code Profiling 8.3

Code profiling collects information about routines or lines of code.

There are two code profiling techniques; each has advantages and disadvantages:

- Timing/Counting. This technique measures the time spent in a routine or counts the number of times a routine is executed.
  - Advantage: Gives very accurate information.
  - Disadvantage: Can significantly slow program execution.
- Sampling. This technique samples the CPU's instruction pointer periodically and records which routine the pointer is in.
  - Advantage: Far less intrusive; the performance overhead is imperceptible.
  - Disadvantage: Sampled data, by its nature, is less accurate.
  - Disadvantage: The technique can tell you *where* time is being spent, but not *why*. For instance, if your program spends most of its time in

the file system, the samples themselves will all be in the file system. You won't know which of your routines is responsible for the file system calls.

As illustrated by the examples later in this chapter, you'll probably want to use each of these techniques under different circumstances.

Code Profiling Options 8.3.1

The syntax for specifying a code profile is:

[*taskSet*] **profile** [*flags*] *routineSet*

The *taskSet* defaults to all tasks.

The *flags* determine what gets recorded in each bucket *and* what profiling technique is used. The flags are one of the following:

+/– *t* Timing. The breakpoint measures how much time is spent in the routine.

+/– *c* Counting. The breakpoint counts the number of times the routine is entered. This is the default.

+/– *h* Sampling. The breakpoint records how often the CPU's instruction pointer is in the routine.

Both +*t* and +*c* can be on in a single profile.

The default flags are +*c* –*t* –*h*.

The *routineSet* determines the buckets that are created for a code profile; DB collects information for each bucket. A routine set is a set of routines or line numbers to be profiled. There are several ways to specify a routine set, as described in the following table.

Technically, a **routineSet** is a set of code addresses, as described in Section 7.2.3.

Table 8-1
Routine Set Specification

Entry	Explanation
scopeSpec	A scope specification for an installed .EXE, .DLL or .OBJ file. The scope specification may be followed by one of the following:
publics	One bucket for each public routine.
statics	One bucket for each static routine.
all	One bucket for each routine. (This is the default.)
lines	One bucket for each executable line.
exes	One bucket for each installed .EXE and .DLL[1].
routineIdentifier	One bucket created for the routine identified with a Scope.Identifier reference, as discussed in Section 7.2.6.
lines	If followed by the optional :lines, then one bucket is created for each line in the routine.
routineSet +/– routineSet	Union or difference of two routine sets.
	* For sampling profiles only, one bucket is created for each code address encountered.

1. Technically, one bucket is created for each code segment of each installed .EXE and .DLL. But it is very rare for an .EXE or .DLL to have more than one code segment.

A Caveat Concerning Sampling Profiles 8.3.2

If DB does not have full symbols for a module being code-profiled, it only knows the start of each public function. Thus if you use Important

```
profile +h "foo.exe"
```

and your source looks like this:

```
STATUS A(void) {...}
static STATUS B(void) {...}
STATUS C(void) {...}
static STATUS D(void) {...}
STATUS E(void) {...}
```

The samples found in B and D will be charged to A and C. If you use

```
profile +c "foo.exe"
```

B and D will be ignored altogether.

You can use these techniques to reduce the impact of this problem:

- When using sample-based code profiles be sure that the modules you profile were compiled with the /D2 flag and were linked with the `DEBUG ALL` line. Code compiled and linked in this fashion provides DB with the start addresses of all the routines, public or private, so DB is much more likely to attribute a sample to the proper routine.
- Use the `STATIC` and `LOCAL` keywords (defined in GO.H) rather than the C language construct static, and compile your code with the `DEBUG` pre-processor variable defined. Under these conditions, the `STATIC` and `LOCAL` routines will actually be public routines rather than private routines. You can then compile with /D1 and link with `DEBUG ALL`. Now DB will have the start location of every public and pseudo-static routine.

Getting More Frequent Samples 8.3.3

You can control the sampling rate. Decreasing the times between samples will increase accuracy but will increase the profiling overhead. The default rate is one sample per 55 milliseconds. You can set that to, say, one sample per 11 milliseconds (probably the smallest reasonable value) by typing:

```
> ?_SetSystick(11)
```

Code Profiling Examples 8.3.4

Sampling Profiles: A First Step 8.3.4.1

To get a rough idea of what some operation is spending its time in, create a profiling breakpoint as follows:

```
>profile +h exes
>g
```

Now perform the operation one or more times. Remember that the sampling rate is a bit coarse (approximately 55 milliseconds), so you want a reasonable number of samples. Finally, get back to DB by typing `PAUSE` and type the following, which lists the time spent in each module, as estimated from the samples taken:

```
>dp 1
```

Refining the Profile with Smaller Buckets 8.3.4.2

Suppose you've found that all your time is spent in the `"foo"` module. You can set up a sampling profile with one bucket per function by saying:

```
>sym "foo"
>profile +h "foo"
```

Or, for just the functions in FILE1.OBJ:

```
>profile +h "foo"."file1.obj"
```

Or, for every source line in FILE1.OBJ:

```
>profile +h "foo"."file1.obj":lines
```

Or, for every source line in the function *SomeFunction*:

```
>profile +h "foo".SomeFunction:lines
```

Refining the Profile with Infinite Buckets 8.3.4.3

If you use the following profile, DB will record every sample, and the **dp** command will then group the samples by function (where it knows symbols) and by .EXE and .DLL (where it doesn't).

```
>profile +h *
```

Timing/Counting Profiles 8.3.4.4

With the following profile, DB will count the number of times each public routine in module `"foo"` is executed. (Notice the *+c* rather than the *+h* used in the previous examples.)

```
>profile +c "foo"
```

With the following profile, DB will accumulate the time spent in each public routine in module "foo". (Notice the *+t* rather than the *+h* and *+c* used in the previous examples.)

```
>profile +t "foo"
```

DB can accumulate both time and count information: The **dp** command will then show you how much time was spent per call.

```
>profile +t +c "foo"
```

With the following, DB will accumulate the time spent in each line in module "foo."

```
>profile +t "foo":lines
```

If specific routines seem to be using the most time, you can add line-by-line hit counts for them by specifying:

```
>profile "foo":all + "foo".RoutineA:lines + "foo".RoutineB:lines
```

Object Profiling 8.4

Object profiling collects information about messages sent via **ObjectCall**.

Basic Object Profiling 8.4.1

The default object profile records and counts all messages sent to all objects:

```
>profile  oc *
```

Object Profiling Options 8.4.2

The general syntax for creating an object profile is:

[*taskSet*] **profile** [*flags*]
oc messagePattern

Notice the *-oc*; this flag tells DB to create an Object profile rather than a Code profile.

The following flags control the number of buckets:

+/– *msg* One bucket per message encountered. (This is the default.)

+/– *msgCls* One bucket per class of message encountered.

+/– *obj* One bucket per receiver object encountered.

+/– *objCls* One bucket per class of message receiver encountered.

The following flags control what gets recorded in each bucket:

+/– *c* Counts the number of messages sent. (This is the default.)

+/– *t* Records execution time.

The default flags are *+msg +c.*

Object Profiling Message Pattern 8.4.3

The message pattern defines what messages are of interest; only messages which match the pattern are recorded. There are several ways to specify a message pattern, as described in the following tables.

Table 8-2
Message Pattern Specification

Entry	Explanation
msgList	Any message in the list matches the pattern.
to objectList	Any message sent to any object in the list matches the pattern.
msgList to objectList	Any message in the list sent to any object in the list matches the pattern.
messagePattern +/- messagePattern	Union or difference of two message patterns.

A *msgList* is specified as follows:

Table 8-3
msgList Specification

Entry	Explanation
*	Any message.
msgFoo	The message with the string name msgFoo.
clsFoo	All messages defined by the class clsFoo.[1,*]
C expression	The message that has the value of C expression.
C expression:msgs	All messages defined by the class that has the value of C expression.[2]
msgList +/- msgList	Union or difference of two message lists.
[msgList]	A message list in required square brackets.

1. Technically, all messages msgX for which ClsNum(msgX) == ClsNum(clsFoo).
2. Technically, all messages msgX for which ClsNum(msgX) == ClsNum(C expression).

An ***objectList*** is specified as follows:

Table 8-4
objectList Specification

Entry	Explanation
*	Any object.
objectName	The object with the string name objectName. (Normally these will be well-known objects.)
clsFoo:objs	All objects of class clsFoo or a subclass of clsFoo.
C expression	The object whose uid has the value of C expression.
C expression:objs	All objects whose class (or some superclass) is the class that has the value of C expression.[1]
objectList +/ objectList	Union or difference of two object lists.
[objectList]	An object list in required square brackets.

1. Technically, call objects objX for which ObjectCall(msgIsA, objX, expression) == stsOK.

Object Profiling Examples

8.4.4

The following command records all **ObjectCalls** to any object that **IsA clsWin**:

```
>profile  oc clsWin:objs
```

The following records all **ObjectCalls** to **clsWin** itself:

```
>profile  oc clsWin
```

The following records all messages in which the class of the message is **clsWin**:

```
>profile  oc clsWin:msgs
```

The following records all messages in task 0508 in which the class of the message is **clsWin** or **clsButton**:

```
>0508 profile  oc clsWin:msgs + clsButton:msgs
```

Displaying Profile Information: the DP Command 8.5

The **dp** command displays the current profile information for a profile breakpoint.

dp [*flags*] *breakpoint*

flags Described below.

breakpoint DB's breakpoint number as displayed by the **bl** command.

The *flags* specify what columns get printed, and their order determines how the rows are sorted. *flags* are a sequence of the following:

Table 8-5
DP Flags

Entry	Explanation
+/– a	code address (code profiles only)
+/– c	count
+/– cpct	displays the percent of the total accumulated hits
+/– fns	group all addresses by functions (useful with routine sets that include :lines or *, if symbolic debugging information is loaded)
+/– h	systick time (code profiles only)
+/– msg	message
+/– msgCls	message class
+/– obj	object
+/– objCls	receiver object class
+/– t	time
+/– tpc	time per count
+/– tpct	like +t but displays the percent of the total recorded times (time in function divided by time in all profiled functions).
+/– tpctt	like +tpct but displays the percent of the total runtime of the task(s) being profiled (time in function divided by time executing anywhere).
+/– z	include entries with a 0 value.

The default flags display everything that was recorded and sort by time (or by time/count if that can be computed).

DP Command Examples 8.5.1

To display the information stored in profile breakpoint 1, type the following. (This profile was collected by tapping three number buttons on the Calculator.)

```
>0508 profile  oc clsWin:msgs + clsButton:msgs
>g
>dp 1
     #hits msg
        18 msgWinGetMetrics
        12 msgWinBeginPaint
        12 msgWinEndPaint
        12 msgWinSend
         9 msgWinGetFlags
         9 msgWinUpdate
```

```
          6 msgWinBeginRepaint
          6 msgWinEndRepaint
          6 msgButtonNotify
          3 msgWinDirtyRect
          3 msgButtonNotifyManager
          3 msgWinRepaint
          3 msgWinSetLayoutDirty
----------
        102
```

Here are several more examples of displaying the same profile information with several different flags.

```
>profile +c +t "calceng"
>g
>dp 1
 #secs/hit          #seconds      #hits addr
   .007115           .099610         14 CalcEngineProcessKey.@443
   .006520           .045646          7 CalcEngineEnterOperator.@377
   .005877           .041145          7 CalcEngineAppendCharToToken.@200
   .005833           .011666          2 CalcEngineEnterNumber.@359
   .002379           .014274          6 CalcEngineSaveValue.@166
   .002083           .006250          3 CalcEngineEvalUnary.@119
   .001733           .015601          9 CalcEngineUpdateToken.@183
   .001546           .001546          1 CalcEngineEvalBinary.@76
            ----------------- ----------
                     .235741         49
>dp +t 1
          #seconds addr
           .099610 CalcEngineProcessKey.@443
           .045646 CalcEngineEnterOperator.@377
           .041145 CalcEngineAppendCharToToken.@200
           .015601 CalcEngineUpdateToken.@183
           .014274 CalcEngineSaveValue.@166
           .011666 CalcEngineEnterNumber.@359
           .006250 CalcEngineEvalUnary.@119
           .001546 CalcEngineEvalBinary.@76
-----------------
           .235741
>dp +tpct 1
  %hits addr
 42.253 CalcEngineProcessKey.@443
 19.363 CalcEngineEnterOperator.@377
 17.453 CalcEngineAppendCharToToken.@200
  6.618 CalcEngineUpdateToken.@183
  6.055 CalcEngineSaveValue.@166
  4.948 CalcEngineEnterNumber.@359
  2.651 CalcEngineEvalUnary.@119
   .655 CalcEngineEvalBinary.@76
```

To display the time sorted by address:

```
>dp +a +t 1
```

To include zero times:

```
>dp +a +t +z 1
```

Clearing Profile Information: the ZP Command 8.6

The **zp** command clears profile data for a profile breakpoint. This allows you to zero out any accumulated profiling information and then start another run, collecting another profile.

zp *breakpoint*

breakpoint DB's breakpoint number as displayed by the **bl** command.

Profiling Specific Tasks 8.7

By default, a profile records the execution of all tasks. You can limit profiling to a task by specifying the task ID:

```
>01228 profile ...
```

You can also limit profiling to all tasks executing a specific application:

```
>"someapp" profile ...
```

Chapter 9 / Advanced DB Techniques

This chapter discusses some advanced or specialized debugging techniques and the DB features that support them. Most of the material in this chapter should be needed rarely, if ever, by a typical PenPoint developer.

Skipping Execution 9.1

DB can skip over pieces of code. This is useful if you know that some piece of code is buggy, and you'd like to skip over it rather than execute it.

In this example, assume that the utility routine called in line 437 will cause a fatal problem:

```
415   MsgHandler (TEAppCaseCmd)
416   {
417       STATUS                    s = stsOK;
418       U32                       gType;
419
420       Dbg(Debugf("Case command self=0x%lx pArgs=0x%lx",self,pArgs);)
421
422
423       switch ((TE_CASE_STYLE)(U32) pArgs) {
424           case teCaseStyleToUpper:
425               gType = xgsRightUpFlick;
426               break;
427           case teCaseStyleToLower:
428               gType = xgsRightDown;
429               break;
430           case teCaseStyleToInitialCaps:
431               gType = xgsRightUp;
432               break;
433           default:
434               s = stsBadParam;
435       }
436       if (s == stsOK) {
437           s = SomeUtilityRoutine(gType, self);
438       }
439       DbgTEAppCaseCmd(("returns 0x%lx",(S32)s))
440       MsgHandlerParametersNoWarning;
441       return s;
442   }
```

We can skip the execution of that routine with the following breakpoint:

```
> bp "minitext.exe"."teusr.obj".@420 {
}   !?EIP="teusr.c".@439;
} }
```

Note that this breakpoint is set on line 420, not on the entry to the routine, so that the initialization of the status variable will occur.

The breakpoint code says, in effect, that when the breakpoint is hit, that the instruction point (EIP) should be set to line 439, and that execution should continue.

Controlling Threads of Execution 9.2

You can control multiple execution threads so that background tasks will not be able to change variables while you trace execution in a selected task. This is useful when debugging programs that have a significant amount of multi-processing in them.

There are three commands that control threads of execution:

The FZ ("freeze") Command 9.2.1

The fz command increments the freeze count on the specified task(s). A task will not execute while its freeze count is greater than 0.

[*taskSet*] **fz**

taskSet A task set as defined in Section 7.2.7. Defaults to the current task.

The TH ("thaw") Command 9.2.2

The **th** command thaws (decrements) the freeze count of the specified task(s). If the freeze count of the task becomes 0, the task is able to run.

[*taskSet*] **th**

taskSet A task set as defined in Section 7.2.7. Defaults to the current task.

The TT ("terminate task") Command 9.2.3

The **tt** command terminates a task or tasks.

[*taskSet*] **tt**

taskSet Set of tasks to be terminated. Defaults to the current task.

Commands Executed at Compile Time 9.3

When a piece of code is attached to a breakpoint (or any DB event handler), the code is compiled into an internal form for later execution.

Some DB commands execute as they are being entered into a piece of code. Therefore these commands are not available for execution inside of a breakpoint's attached code.

The DB commands that execute at compile time are:

<	log
h	mini
id	qsrcdir
ids	type
files	vars
fns	ver

DB Built-ins

9.4

For advanced use, DB has many useful built-in types, variables, and routines.

DB's Predefined Types

9.4.1

Table 9-1
DB's Predefined Types

Type	Description
BOOLEAN, P_BOOLEAN	as in go.h
S8, P_S8	as in go.h
U8, P_U8	as in go.h
CHAR, P_CHAR	as in go.h
S16, P_S16	as in go.h
U16, P_U16	as in go.h
S32, P_S32	as in go.h
U32, P_U32	as in go.h
SIZEOF	as in go.h
CONTEXT	as in go.h
P_ARGS	as in go.h
_HEX_LONG	a U32 that the ? command displays in hex
_HEX_SHORT	a U16 that the ? command displays in hex
_HEX_BYTE	a U8 that the ? command displays in hex
_BITS	an enum with values b0=1, b1=2, b2=4, etc
MESSAGE, MSG	a U32 that the ? command displays as a message name
OBJECT, OBJ	a U32 that the ? command displays as an object name
STATUS, STS	a U32 that the ? command displays as a status name
_MSG_PAT	a message pattern; see below.
_ROUTINE_SET	a routine set; see below.
_TASK_SET	a task set; see below.

The last three types can be used to declare variables that can be set and then used as parameters to certain DB commands. This is particularly useful because values of these types can potentially be very complex.

Here's an example of a _MSG_PAT variable being used as a parameter to the **profile** command:

```
{
  _MSG_PAT mp1 = [mp clsWin:msgs];
  profile +c  oc (mp1);
}
```

Here's an example of a _ROUTINE_SET variable being used as a parameter to the **profile** command:

```
{
  _ROUTINE_SET rs1 = [rs "winc"+"win0":lines];
  profile +c (rs1);
}
```

Here's an example of a _TASK_SET variable being used as a parameter to the **tl** command:

```
{
  _TASK_SET ts1 = [ts "minitext"+"browser"];
  [(ts1)] tl;
}
```

Useful Values in DB 9.4.2

The values in the following table are occasionally useful. In general, you should not set the values.

Table 9-2
Useful Values in DB

Type	Name	Description
		Values Always Available
TASK_SET	_allTS	The task set containing all tasks.
TASK_SET	_allProgTS	The task set containing all applications (that is, all processes running .exes).
U32	_nTicks	The total elapsed systicks.
_RUNTIME	_runtime	The total run time over all tasks.
		Values Available for the Current Task
_T_REGS	regs	The current task's registers, as a structure.
_HEX_LONG	eip, eax, ebx,...	The current task's registers, individually.
_FLAGS	efg	The current task's flags.
_HEX_LONG	esp0	The current task's ring-0 stack pointer.
_HEX_SHORT	cs	The current task's code-segment register.
OS_TASK_ID	processId	The current task's process id.
U16	instanceNum	The current task's instance number.
		Values Available Within an Event Handler
OS_TASK_ID	_taskId	The victim task; irrelevant in an "on intReq" handler.
_TASK_FRAME	_taskFrame	The resume address and top frame pointer of <_taskId>.
_HANDLER_NUMBER	_handlerN	The handler number of the current handler.
_P_UNKNOWN	_pData	The handler data of the current handler.
_PARSED_PROG_NAME	_ppn	If "on installBegun", "on installDone," "on start," the name of the program, parsed into its components: vendor-program-major(minor).
		Values Available Within an OC Event Handler
MESSAGE	msg	If "on oc," the message.
OBJECT	self	If "on oc," the message recipient.
P_ARGS	pArgs	If "on oc," a pointer to the message arguments.

DB's Useful Variables

9.4.3

The variables in the next table can be examined and set.

Here's an example of setting the **_keepTypedefs** variable:

```
>!_keepTypedefs = true;
```

Table 9-3
DB's Useful Variables

Type	Name	Default	Description
BOOLEAN	_showRuntime	false	If true, g, t, p, etc., show the task runtime.
BOOLEAN	_showRealtime	false	If true, g, t, p, etc., show the elapsed time.
BOOLEAN	_suspCur	true	DB suspends the current task while in DB command mode; if false, DB lets it run.
BOOLEAN	_redirectInput	true	When not in command mode, DB redirects input to the PenPoint input system; if false, such input is just DB typeahead.
BOOLEAN	_redirectPrinting	false	If true, remote DB redirects target Debugf() output to the host.
BOOLEAN	_keepUnusedTypes	false	DB discards any types from an .exe/dll that are not actually "reachable" from some function or variable definition.
BOOLEAN	_keepTypedefs	false	If false, DB discards any typedef definitions from an .exe/.dll.
_TASK_SET	_starTS	_pAllProgTS	The task set you get when you say "[*]", and the default task set for commands like bp, tl, etc.
_TASK_SET	_suspTS	empty set	DB suspends all these tasks (in addition to the current task) whenever it enters command mode.
S8	_defCodeMode	1 ('l')	Mode of u command. Currently the only legal value is 1.
U16	_defCodeLength	8	Number of lines output by u command.
S8	_defViewMode	0 ('a')	Mode of v command. May also be –1 ('b') or 1 ('l').
U16	_defViewLength	8	Number of lines ouput by v command.
S8	_defDataMode	1 ('l')	Mode of d, db, dw, and dd commands. May also be –1 ('b') or 0 ('a').
U16	_defDataLength	64	Length (in bytes) for d, db, dw, and dd commands.
S8	_defShowStackMode	0 ('a')	Mode for st command. May also be 1 ('l') or –1 ('b').
_SHOW_STACK_FLAGS	_showStackFlags	ssParams	Any OR'd combination of: _ssParams (show call parms), _ssLocals (show block locals), _ssCtx (show current execution address of stack frame).

continued

Table 9-3 (continued)

Type	Name	Default	Description
U32	_maxCachedBytes	128K	DB swaps some data between memory and the file DBSWAP (e.g. declaration lists, line numbers). _maxCachedBytes is the memory cache limit.
U16	_defShowStackLength	8	Number of frames shown by st command.
S32	_defDumpObjectParam	-1	P_ARGS sent by od command.

DB Runtime Routines

9.4.4

The following routines are available in the DB runtime.

As in the C runtime:

memcpy

printf

putchar

sprintf

strlen

As in CLSMGR.H:

ObjectCall

ObjectCallAncestor

ObjectSend

ObjectSendUpdate

ObjectPost

ObjectPostAsync

Returns the debug flags of set <set>.

```
_HEX_LONG _GetFlags(U16 set)
```

Sets the debug flags <set> to <flags>.

```
void _SetFlags(U16 set, _HEX_LONG flags)
```

Delays DB execution for <ms> milliseconds.

```
void _Delay(U32 ms)
```

Sets the systick rate, in milliseconds.

```
void _SetSystick(U32 ms)
```

Does a `printf` and then raises a DB error, typically forcing DB to command mode.

```
void _Err(CHAR * pFmt, ...)
```

Allocate globally accessible memory.

```
void * _Alc(SIZEOF size)
```

Free globally accessible memory.

```
void _Free(void * pMem)
```

The ON Command 9.5

The **on** command creates a handler for specified events.

In general, the syntax for the **on** command is:

[*taskSet*] **on** *event* [*eventAction*]

taskSet A task set as described in Section 7.2.7. Defaults to all tasks.

eventAction C code to be executed when the breakpoint is hit. The default action is to break to the debugger.

There are several categories of events.

Program Events 9.5.1

An installBegun event happens when instance 0 of a program begins execution.

```
[taskSet] on installBegun defNameString  [eventAction]
```

An installDone event happens when instance 0 of a DLL finishes executing **DLLInit**().

```
[taskSet] on installDone defNameString [eventAction]
```

A start event occurs when any instance of a program besides instance 0 begins execution.

```
[taskSet] on start defNameString [eventAction]
```

Access Events 9.5.2

An access event occurs when the variable specified by the lvalue is read or written.

```
[taskSet] on access lvalue [eventAction]
```

A store event occurs when the variable specified by the lvalue is written.

```
[taskSet] on store lvalue [eventAction]
```

Task Events 9.5.3

A tick event happens every systick interval.

```
[taskSet] on tick [eventAction]
```

A step event is an instruction step

```
[taskSet] on step [eventAction]
```

A pstep event is the same as a step event but skips over CALLs.

```
[taskSet] on pstep [eventAction]
```

A newST event is the creation of a new subtask;

```
[taskSet] on newST [eventAction]
```

A newIST event is the creation of a new interrupt subtask

```
[taskSet] on newIST [eventAction]
```

A terminate event is the termination of a task.

```
[taskSet] on terminate [eventAction]
```

A debReq event happens when a task calls **Debugger**().

```
[taskSet] on debReq [eventAction]
```

Fault Events 9.5.4

A fault event occurs whenever a hardware fault occurs. The special event fault is shorthand for ORing all of the fault events

xxxxxx

```
[taskSet] on div [eventAction]
[taskSet] on nmi [eventAction]
[taskSet] on into [eventAction]
[taskSet] on bound [eventAction]
[taskSet] on invOp [eventAction]
[taskSet] on extNA [eventAction]
[taskSet] on invTSS [eventAction]
[taskSet] on segNP [eventAction]
[taskSet] on stkOv [eventAction]
[taskSet] on gp [eventAction]
[taskSet] on unk [eventAction]
[taskSet] on fault [eventAction]
```

Other Events 9.5.5

An exit event happens when a block of interpreted C code is exited. Exit events are used to establish cleanup code.

```
{   <initialization code>
    !on exit { <cleanup code> }
    <code that may exit>
}
```

An error event happens when an error occurs during execution of interpreted C code.

```
{   !on error { printf("BAD ADDRESS\n"); return; }
    *pData = 0;
}
```

An intReq event happens whenever DB's break key is pressed.

```
on intReq [eventAction]
```

A bp event occurs whenever the codeAddress is executed.

```
[taskSet] on bp codeAddress [eventAction]
```

An oc event occurs whenever a message matching the message pattern is delivered.

```
[taskSet] on oc messagePattern [eventAction]
```

The INSTALL and START Commands 9.6

DB's **install** command does an **OSProgramInstall()** of an .EXE or .DLL.

DB's **start** command does an **OSProgramInstantiate()** of an .EXE or .DLL.

These commands cannot be used to test PenPoint applications because they do not do all of things that the Application Framework does when installing and instantiating a PenPoint application.

Context Inside of Breakpoints 9.7

When DB executes code inside of a breakpoint, it does not set the global "current source position" information. (This is done so that it is possible to write breakpoint code that doesn't disturb that part of the global context.) This means that the source viewing commands (**v**, **u**, **vu**, and **uv**) may not display what you expect.

To set the current source position when executing code inside a breakpoint, do the following in the breakpoint code:

```
! . ctx
```

Here's a more complete example:

```
> bp SomeFunction {
}   ! . ctx;
}   ! v a 5;
}   ...
}   }
```

Cast Operator 9.8

A cast may be used as a postfix operator. For example, `x(int)` is equivalent to `(int)x`.

Tilde Operator 9.9

The tilde can be used as a postfix dereference operator. For example, `x~` is equivalent to `*x`.

Chapter 10 / General PenPoint Debugging Techniques

This chapter describes the other tools and techniques that you can use to debug PenPoint programs. Many of these tools or techniques use three facilities provided in the PenPoint™ operating system:

- The DEBUG compiler option, which causes specific message passing macros to send text to the debugger stream.
- Debugger flags, which you can use to turn on and turn off certain behaviors in a program.
- The debugger stream, a pseudo device to which programs can write text. You can view the debugger stream either on a second monitor or in the System Log application, which is described in Chapter 11. After shutting PenPoint down, you can view the debugger stream in a log file, named PENPOINT.LOG.

Unfortunately these topics are tightly interrelated, it is hard to describe one without requiring some knowlege of the other. If you have questions while reading this chapter, keep reading; your question will probably be answered later on.

DEBUG Compiler Option 10.1

When you compile an application, you can use the /D compiler switch to create the preprocessor `#define` name DEBUG.

```
set WCC386=/3s /Oif+ /s /W3 /We /Zc /Zq /fpc /D2 /En /DDEBUG
wcc386p /Foemptyapp.obj emptyapp.c > emptyapp.err
```

PenPoint Uses DEBUG 10.1.1

When you specify the DEBUG `#define` name in compilation, several PenPoint macros in DEBUG.H, GO.H, and CLSMGR.H are defined so that they send information to the debugger stream (**StsWarn, StsFailed, StsRet, StsJmp**, and **StsOK** in GO.H, **ASSERT** in DEBUG.H, and a number in CLSMGR.H). See the tables at the end of this chapter for a complete listing.

StsWarn evaluates any expression that returns STATUS. If STATUS is an error, **StsWarn** writes the status value, the file name, and line number to the debugger stream. The message passing macros, **StsFailed, StsRet, StsJmp**, and **StsOK** all use the **StsWarn** macro.

You can use the macro **Dbg** to comment out single line debug statements when DEBUG is not defined. For example:

```
Dbg(Debugf("Only shows up in DEBUG version");)
```

Often when working on functions called by other functions, you assume that the software is in a certain state. The **ASSERT** macro lets you state these assumptions, and if DEBUG is set, it checks to see that they are in fact the case. If the assumptions are not satisfied, **ASSERT** will send text to the debugger stream. For example, a square root function might rely on never being called with a negative number:

```
void MySqRoot(int num)
{
    ASSERT(num >= 0, "MySqRoot: input parameter is negative!");

    // Calculate square root...
```

If DEBUG is defined and the assertion is not true, **ASSERT** will send "MySqRoot: input parameter is negative!" to the debugger stream.

Using DEBUG in Your Programs 10.1.2

You can also test for the DEBUG `#define` name in your programs. If you include `#ifdef` and `#endif` statements to test for DEBUG, you can make your application behave differently when it is compiled with DEBUG. For example:

```
#ifdef DEBUG
    DPrintf("Pen Moved to location (%ld,%ld)\n", pArgs->xy);
#endif
```

The function **DPrintf()** is described later in this chapter.

Debug Versions of PenPoint DLLs 10.1.3

The PenPoint SDK includes debugging versions of the PenPoint DLLs. These print out informative errors and use the DEBUG versions of STATUS checking macros.

To use the debugging versions of system DLLs, modify \PENPOINT\BOOT\BOOT.DLC to load the debugging versions. For instance, change this line:

```
go-input-v1(0)       \\boot\penpoint\boot\dll\input.dll
```

to this:

```
go-input-v1(0)       \\boot\penpoint\boot\_dll\input.dll
```

Debugging Flag Sets 10.2

PenPoint sets aside 256 variables, called **debugging flag sets**, that you can use when debugging. Each variable is a 32-bit value, which means that you can assign at least 32-different meanings to each debugging flag set.

Because there are 256 debug flags sets, they can be indexed by an 8-bit character. Commonly, we refer to a specific debugging flag set by the character that indexes that flag. GO has reserved all the uppercase character debug flags sets (A through Z), and has reserved some of the lowercase characters also. To find which debug flags set are available, see the file \PENPOINT\SDK\INC\DEBUG.H.

You can set the value of a debugging flag set and retrieve it. There are several ways to set values in debugging flag sets:

- Using the function **DbgFlagSet()** in a program.
- Using the *debugset* environment variable in \PENPOINT\BOOT\ENVIRON.INI.
- Using the **fs** command in either DB or the mini-debugger.

Your application can determine the value of a debugging flag set with the function **DbgFlagGet()**.

Typically, you set a flag value before running a program and your program uses **DbgFlagGet()** to check the value of the flag.

Do not use these in production code; it is fine to use them in DEBUG versions, but they are not administered and could cause hard-to-find errors.

Some PenPoint modules include additional code that can be activated via these debugging flag sets. Header files describe the effects of setting different bits in each debugging flag set. For example, setting the 4000 bit in C causes the debugging version of the Class Manager to print miscellaneous warning messages.

The Debugger Stream 10.3

As described in the introduction to this chapter, the debugger stream is a pseudo device to which programs can write text.

There are four different ways to view the information sent to the debugger stream:

- If logging is turned on, all information written to the debugger stream is written to the log file.
- You can direct the debugger stream to send its information to a serial port.
- If you have two monitors (VGA and monochrome), the debugger stream is displayed on the second (monochrome) monitor.
- If you have the System Log application running, the debugger stream is captured by the System Log.

These destinations are not mutually exclusive. If all three are active, all three will receive the debugger stream data. The System Log application is described in detail in Chapter 11.

Configuring the Debugger Stream Destinations 10.3.1

You can control where the debugger stream is routed by modifying lines in your MIL.INI and ENVIRON.INI files. This section describes the various debugger stream destinations and how you configure them. See the next chapter (Chapter 11) for information on using System Log Application.

Writing to a Log File 10.3.1.1

To direct the debugger stream to a log file, set the debugger flag D to the value 8000. There are several ways to do this:

- In DB or the mini-debugger, use the fs command: fs D 8000.
- In ENVIRON.INI, add the DebugSet line: DebugSet=/DD8000.

The _ENVIRON.INI file that is shipped with the SDK contains the DebugSet line, but it is commented out. The simplest way to enable logging is to remove the pound sign (#) comment marker at the beginning of the line

```
#DebugSet=/DD8000
```

By default, the debug output is written to the file \\PENPOINT.LOG. To change the destination of the log file, add a DebugLog line to your ENVIRON.INI file. For example, these lines cause the debugger stream to be logged in the file MY.LOG in the root of the system volume:

```
DebugSet=/DD8000
DebugLog=\\MY.LOG
```

Normally output to the debug log is buffered, to improve performance. However, this can cause problems if the machine crashes before it can write the last buffered data to the log file. To avoid this, you can add a DebugLogFlushCount line to your ENVIRON.INI file. This line specifies the number of characters that can be written to the buffer before the buffer is flushed to the log file.

Writing to a Serial Port 10.3.1.2

You can direct the debugger stream to write to a serial port. You can then attach a dumb terminal (or a PC running a terminal emulation package) to the serial port and view the debugger stream, much like using a second monitor.

This is most useful when you are debugging a PenPoint program on a PenPoint computer and have no other way to view the debugger stream, other than the PENPOINT.LOG file.

To write debugging information to a serial port, edit your MIL.INI file and add a SerialDebugPort line to it. The _MIL.INI file that is shipped with the SDK contains two SerialDebugPort lines already configured, but commented out.

```
# Remove the "#" on the next line to route debugging information to COM1
#SerialDebugPort=1
# Remove the "#" on the next line to route debugging information to COM2
#SerialDebugPort=2
```

Writing to a Second Monitor 10.3.1.3

Developing PenPoint applications is much more pleasant if you have a two-monitor configuration (one VGA, one monochrome). With two screens you can interact with your program while watching the debugger stream on the monochrome screen. With only one monitor, you can only view the debugger stream when you halt PenPoint (with the Pause key).

If PenPoint is running on a PC and it detects a second (monochrome) monitor, it will display the debugger stream on that monitor. All you have to do is add a monochrome card to your machine and attach a monochrome monitor to the card.

If, for some reason, you want to turn off the debugger stream from the second monitor, add the line `MonoDebug=off` to your MIL.INI file. The _MIL.INI file shipped with the SDK contains a MonoDebug line, but it is commented out.

Viewing on a Single Monitor 10.3.1.4

If you have a single monitor on your PC, you can use the System Log application to view the debugger stream. However, you cannot run the System Log application while PenPoint is booting. To view the debugger stream during the boot sequence, you must:

- Modify the MonoDebug line in your MIL.INI file so that it specifies "off."
- Ensure that your boot volume contains the file \PENPOINT\BOOT\CONSOLE.DLC.

The CONSOLE.DLC file lists screen drivers that display debugging information on the VGA screen during booting.

Writing to the Debugger Stream 10.3.2

In describing other aspects of debugging, we have already described several ways that you can write to the debugger stream. These tables summarize the functions and macros that send text to the debugger stream.

The functions in the Table 10-1 always write text to the debugger stream (that is, they don't have to be compiled with the DEBUG preprocessor `#define` name).

Table 10-1
Explicit Writes to the Debugger Stream

Function	Purpose
Debugf	Writes a formatted string (like printf), with a trailing newline.
DPrintf	Writes a formatted string, like printf, with no trailing newline.

Table 10-2 lists the "warn" form of the PenPoint message passing macros. If you compile these macros with the DEBUG preprocessor `#define` name and the message that they pass returns a status value less than **stsOK**, the macros write a string to the debugger stream. The string contains the name of the file and the line number where the unsuccessful message occurred.

If you compile these macros without DEBUG, they simply call the standard form of the function. For example, without DEBUG **ObjCallWarn** will call **ObjectCall()**.

Table 10-2
Warning Message Passing Macros

Macro	Purpose
ObjCallWarn	Calls ObjectCallWarning().
ObjCallAncestorWarn	Calls ObjectCallAncestorWarning().
ObjCallAncestorCtxWarn	Calls ObjectCallAncestorCtxWarning().
ObjSendWarn	Calls ObjectSendWarning().
ObjSendUpdateWarn	Calls ObjectSendUpdateWarning().
ObjSendTaskWarn	Calls ObjectSendTaskWarning().
ObjSendUpdateTaskWarn	Calls ObjectSendUpdateTaskWarning().
ObjSendU32Warn	Calls ObjectSendU32Warning().
ObjPostWarn	Calls ObjectPostWarning().
ObjPostAsyncWarn	Calls ObjectPostAsyncWarning().
ObjPostTaskWarn	Calls ObjectPostTaskWarning().
ObjPostAsyncTaskWarn	Calls ObjectPostAsyncTaskWarning().
ObjPostU32Warn	Calls ObjectPostU32Warning()

Table 10-2 lists the warning forms of the PenPoint message passing functions. When the message passed by these functions returns a status value less than **stsOK**, these functions write text to the debugger stream.

Usually you don't use these functions but you use the related macros, listed in Table 10-3.

Table 10-3
Message Passing Functions

Function	Purpose
ObjectCallWarning	Calls ObjectCall(), writes if status is less than stsOK.
ObjectCallAncestorCtxWarning	Calls ObjectCallAncestorCtx(), writes if status less than stsOK.
ObjectCallAncestorWarning	Calls ObjectCallAncestor(), writes if status less than stsOK.
ObjectSendWarning	Calls ObjectSend(), writes if status less than stsOK.
ObjectSendUpdateWarning	Calls ObjectSendUpdate(), writes if status less than stsOK.
ObjectSendTaskWarning	Calls ObjectSendTask(), writes if status less than stsOK.
ObjectSendUpdateTaskWarning	Calls ObjectSendUpdateTask(), writes if status less than stsOK.
ObjectPostWarning	Calls ObjectPost(), writes if status less than stsOK.
ObjectPostAsyncWarning	Calls ObjectPostAsync(), writes if status less than stsOK.
ObjectPostTaskWarning	Calls ObjectPostTask(), writes if status less than stsOK.
ObjectPostAsyncTaskWarning	Calls ObjectPostAsyncTask(), writes if status less than stsOK.

You can use these macros when invoking a macro or function that evaluates to a status value. If you compile these macros with the DEBUG preprocessor `#define` name and their expression evaluates to less that **stsOK**, the macros write a string to the debugger stream (and some perform further actions). The string contains the status value and the name of the file and the line number where the unsuccessful expression occurred.

Table 10-4
Expression Handling Macros

Macro	Purpose
StsWarn	If not OK, writes to debugger stream.
StsFailed	If not OK, calls StsWarn.
StsRet	If not OK, calls StsWarn and returns with status.
StsJmp	If not OK, calls StsWarn and jumps to specified location.
ASSERT	If assertion fails, writes to debugger stream.

Chapter 11 / The System Log Application

The System Log application allows you to view text written to the debugger stream. The System Log application reads (but does not intercept) text written to the debugger stream, and saves the text in its internal buffer.

You can scroll the System Log application's contents so that you can review your debugging session. The System Log application preserves an adjustable number of lines of debugging output. The application also has several menu commands that give you quick access to memory information, enable debug logging, and get and set debug flags.

Loading the System Log Application 11.1

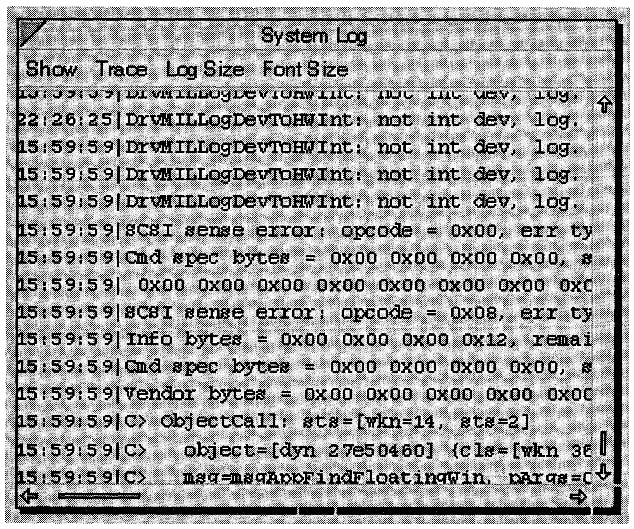
The System Log application is a normal PenPoint application. You can configure the PenPoint™ operating system so that the application is loaded at boot time or you can load the application using the application installer.

The line needed to load the System Log application at boot time is already in the SYSAPP.INI file. However, the line is commented out. To load the System Log application at boot time, just remove the pound sign (#) comment at the beginning of the line.

Running the System Log Application 11.2

The System Log application is a accessory. You can run the application by opening Accessories and then tapping on the System Log icon.

Figure 11-1
System Log Application on a PC



You can scroll, resize, move, open, and close the System Log application as you would any other floating accessory. For more information on using tools, see the end-user documentation.

To close the System Log application, tap its close corner. The lines in the System Log application are preserved (and are also in the debug log file if you enable logging).

System Log Application Menus 11.3

The System Log application has four menus that you can use to control the application or display system information. They are: Show, Trace, Size, and Font Size.

Show Menu 11.3.1

The items in the Show menu direct the System Log application to display information about the current system.

Memory Usage Displays the current memory used by PenPoint and application.

Task List Lists all of the tasks currently running.

Device List Lists all devices currently installed.

Trace Menu 11.3.2

The Trace menu allows you to specify the types of messages displayed on the System Log's window.

Application Errors Display all application and non-system errors from the **StdError()** function.

Non-error Messages Display all application and system dialog messages from the **StdMsg()** function.

Off Turns off all debugging messages, including those from **Debugf**, **DPrintf**, and **StsWarn**.

The Off choice only appears in the version of the System Log application that was compiled with the DEBUG preprocessor #define name (in _PP\BOOT\APP). If you can't see the Off choice, you are using the production version of the System Log application from \PENPOINT\BOOT\APP.

Log Size Menu 11.3.3

The log size menu allows you to choose how large a buffer to reserve for the System Log. When the System Log is full, the oldest lines are removed before new lines are added. If you are working with limited amounts of memory, you may want to choose a smaller size for the System Log.

The choices are small, medium, and large. Small is 8K bytes (approximately 6 computer screens of information); medium is 16K bytes; large 32K bytes. The default size is medium.

Font Menu 11.3.4

The font menu allows you to choose the size font to display the system log. Possible sizes are: 8, 10, 12, 14, 18, and 24 point type.

Chapter 12 / PenPoint Mini-Debugger

In addition to **DB**, PenPoint has a much simpler kernel debugger. The mini-debugger is not as full featured as DB. It can't display source code lines or symbols, but it has minimal memory requirements, doesn't require any configuration, and can debug any code (including kernel code).

If the mini-debugger is loaded and any process crashes, you can determine what line your program crashed at, what routines called which routines, and what PenPoint modules were involved. All you have to do is log the results of your mini-debugger session, use the mini-debugger to get some system and process information, and compare the results with the .MAP files produced by the compiler and linker.

The mini-debugger lets you:

- Get information about tasks, heaps, segments, memory usage, the memory file system.
- Perform stack traces.
- Disassemble code and display and modify registers.
- Set breakpoints and step through code at the assembler language level.

The mini-debugger is a part of the PenPoint system, so it is always loaded.

Invoking the Mini-Debugger 12.1

When running the PenPoint™ operating system on a PC, pressing Pause enters the mini-debugger. Its prompt is a greater-than sign, >. If you're running on a single-monitor configuration, the PenPoint graphical screen is replaced by a text screen, otherwise output appears on the second monitor. You can then type commands to the mini-debugger.

PenPoint also enters the mini-debugger whenever a process has a general protection fault or page fault.

Also a program can explicitly enter the mini-debugger by calling the **Debugger**() system routine.

Mini-Debugger and DB 12.1.1

If you have installed DB, then all of the above actions invoke DB instead of the mini-debugger. You can tell the difference between them because the DB > prompt has a "cursor," a dot which indicates the current insertion point. You can enter the mini-debugger from DB by typing **`mini`**, and return to DB from the mini-debugger by typing **g**.

On a PenPoint Computer 12.1.2

When PenPoint enters the mini-debugger, it expects commands from a keyboard. Of course, most PenPoint computers do not normally have a keyboard, so it is difficult to enter a g command to continue.

Therefore, the production version of PenPoint (intended to run on a PenPoint computer) has the D 10000 debugger flag is set. This flag disables the mini-debugger. When PenPoint fails, it dumps the registers to the debugger stream and attempts to continue.

Mini-Debugger Commands 12.2

Typing **?** displays the available mini-debugger commands.

Table 12-1
Mini-Debugger Commands

Command and Parameters	Description
?	Evaluate and print expressions.
ai addr	Display address information.
ap cmd [;cmd]	Execute the commands in all ctx of all processes.
at cmd[;cmd]	Execute the command in all ctx of all tasks.
bc number	Clear Breakpoint.
bd { number | *}	Disable breakpoints.
be { number | *}	Enable breakpoints.
bl number	List breakpoints.
bp number addr	Set breakpoint number at addr.
cd u32	Call objects with msgDump, pArgs=0.
cm u32	Convert msg.
co u32	Convert object.
cs u32	Convert status.
ct u32	Toggle trace setting for objects.
ctx [task-id]	Set the current task.
d [rangeList]	Display memory at address.
da [rangeList]	Display ASCII at address.
db [rangeList]	Display bytes at address.
dd [rangeList]	Display double words at address.
dw [rangeList]	Display words at address.
fd fileName	Display file.
fl [flag]	Display debug flags.
fs flag [value]	Set/Clear debug flags.
g [addr]	Continue program execution (until address).
hd addr	Display heap summary and allocated blocks information.
help [cmdName]	Display help.
hi addr	Heap status.
hl	Display heaps in process.

continued

Table 12-1 (continued)

Command and Parameters	Description
inb port	Input byte from I/O port.
inw port	Input word from I/O port.
ldev	Display MIL devices.
mdb controls	Special Debugger control settings.
mi	Display task memory information.
outb port value	Output byte to I/O port.
outw port value	Output word to I/O port.
p [physaddr]	Display physical memory.
q [exitCode]	Exit PenPoint.
qq	Exit PenPoint, skip standard shutdown.
r	Display standard registers.
ra	Display all registers.
rf	Display floating point registers.
rx	Display special registers.
s [repeatCount]	Step – Single step.
si [semaphore]	Display semaphore information.
st [addr [n]]	Stack trace at address for n frames.
t [repeatCount]	Trace—single step.
ti	Task information.
tl [taskList]	Display task list.
tt task	Terminate this task.
tst taskId [addr]	Stack Trace of another task.
u [rangeList]	Disassemble code.
w	Warm boot (PC only).

Setting Debug Flags 12.2.1

You use the **fs** command to set debug flags. There are two forms of the command:

fs *c hhhhhhhh*

where **c** is any one byte ASCII character that identifies the debug flag set, and hhhhhhhh is a hexadecimal value of 1 to 8 digits. There is no space between the character and hexadecimal value.

fs *hh hhhhhhhh*

where **hh** is a one or two digit hexadecimal value that specifies the debug flag set, and **hhhhhhhh** is a hexadecimal value of 1 to 8 digits. In this form of the `fs` command, spaces are significant. This form allows you to set or clear debug flags that are identified by non-printing ASCII characters (such as those in the ranges 0 through 32 and 128 through 255).

To clear a debugging flag set, set the flag set to 0.

Command	Action
fs *c hhhhhhhh*	Set flag identified by character *c* to hex value *hhhhhhhhh*.
fs *c* 0	Clear flag identified by character *c*.
fs *hh hhhhhhhh*	Set flag identified by hex value *hh* to hex value *hhhhhhhh*.
fs *hh* 0	Clear flag identified by hex value *hh*.
fl *hh*	Display flag identified by hex value *hh*.

For example, either of these two commands sets the F debug flag to 10:

```
fs F 0010
fs 46 0010
```

Using the Mini-Debugger 12.3

If you enter the mini-debugger, you usually:

- Type **fs D 8000** to enable logging (if logging is not already enabled).
- Type **st** to get a stack trace. This should indicate which code the error occurred, and where it was called from.
- If you're familiar with assembly code, type **r** to display registers and **u** to unassemble the code which failed.
- Type **ti** to see the state of tasks.
- Then type **g** to try to continue.

Map Files 12.4

A map file records line numbers and symbol addresses created by the compiler and linker. A map file is useful when decoding faults and interpreting stack traces produced by PenPoint's mini-debugger. The WATCOM Make files in the \PENPOINT\SDK\SAMPLE directories assign the extension .MPE to map files for executable modules, and .MPD to map files for DLL modules.

To display symbolic names for routines from the mini-debugger, compile your application with the /En switch.

Exception Handling 12.5

If PenPoint entered the mini-debugger due to a general protection fault and you enter **g** to continue execution, the kernel general protection fault handler terminates the task that encountered the fault.

Understanding Interrupts 12.6

The best place to start interpreting interrupts is with Intel's *386SX/DX/SL Microprocessor Programmer's Reference Manual.* In Chapter 9, Exceptions and Interrupts, there's an explanation of the interrupts (Section 9.9) and the **errorCode** registers (Section 9.8).

Some of the interrupts are faults that cause PenPoint to crash. For example, fault 12 is a stack fault. Some faults can be handled by interrupt handlers.

Sometimes an interrupt handler will drop to DB. If this happens, the registers printed by the mini-debugger are the registers for the interrupt handler. However, because this isn't very useful, DB tries to be helpful by printing the registers for the task that was active when the interrupt handler kicked in. You need to look at those registers. For example:

```
FAULT 12 in nonpreemptable task 558

fault 12, errorCode 0000, task 0558
ax 0000FF08 bx 00436C8C cx 0000000F dx 000003CE si 40774988 di 4031F17E
sp 0044DF20 bp 00436C8C ip 40774CFD 0---0--I--Z-P-
ax 0000FF08 bx 00436C8C cx 0000000F dx 000003CE si 40774988 di 4031F17E
sp 0044DF20 bp 00436C8C ip 40774CFD 0---0--I--Z-P-
 EAX=40670000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00420000 EDI=00004246
 EIP=40663C85 EBP=00426EE0 ESP=00426EDC FLG=00004246 CR2=40CB1078 CR3=0018F000
 CS=0010 DS=0020 SS=0090 ES=0020 FS=0000 GS=0000 TSS=0088 TNAME=Stak
```

The key set of registers is the first set (lowercase two-letter registers). The capitalized registers are the interrupt subtask calling the debugger. The giveaway is that the task where the fault happened is 558 (a fairly large task ID, so it is probably an application task), but the TSS of the second set is low (probably a system task). The task name (TNAME) is Stak, which makes sense because fault 12 is (from the Intel book) "Stack Exception."

If you immediately do a stack trace or unassemble the code around the instruction pointer, you get the second set:

```
>u eip-20
 40663C65 85DB                TEST    EBX,EBX
 40663C67 740E                JE      40663C77
 40663C69 68D82B6640          PUSH    40662BD8
 40663C6E FF15C0D16640        CALL    [4066D1C0]
 40663C74 83C404              ADD     ESP,04
 40663C77 66C705306467400100  MOV     [40676430],0001
 40663C80 E871F89C9F          CALL    OSDebugger (E00334F6)  <<<<
 40663C85 66C705306467400000  MOV     [40676430],0000
 40663C8E 6685DB              TEST    BX,BX
 40663C91 740E                JE      40663CA1
 40663C93 6814286640          PUSH    40662814
 40663C98 FF15C0D16640        CALL    [4066D1C0]
 40663C9E 83C404              ADD     ESP,04
 40663CA1 E843FFFFFF          CALL    DoReinstallDB (40663BE9)
 40663CA6 6685DB              TEST    BX,BX
 40663CA9 740E                JE      40663CB9
```

All this tells you is that the interrupt subtask dropped into the debugger. What we really want is what was happening that caused the interrupt subtask to mess up, so let's take a look at the whole task list:

```
>tl
Task Id      type parent      state    class  pri  memused   name   time(s)
-------      ---- ------      -----    -----  ---  -------   ----   -------
  48      process             ready      low   50  172032    Idle   20.053

  68    Interrupt     48                                     Dbl     0.000
  78    Interrupt     48                                     TSS     0.000
  88*   Interrupt     48                                     Stak    0.000
  98      process          msg wait   system    9    8192    Scav    0.548
  A8      process          msg wait     high   44    4096    V86x   40.474
  B8    Interrupt     48                                     V86a    0.000
  C8      Subtask     48      ready med high    7            Page    0.847
  D8      Subtask     48   msg wait     high    3            Powr    0.145
  E8      Subtask     48      ready     high   45            DmIM    0.070
  F8      Subtask     48  sema wait  med low    7                    0.000
 108      process          msg wait  med low    7  835584    Syst    8.711
 118      Subtask    108      ready  med low    7            Timr    0.023
 128      Subtask    108  sema wait     high   46            DBm    21.608
 138      Subtask    108      ready     high   45            DBk     0.008
 148      Subtask    108   msg wait  med low    7                    0.359
 158      Subtask    108   msg wait  med low    7            IMgr    0.017
 168      Subtask    108  sema wait  med low    7            IDSP    0.771
 178      Subtask    108      ready     high   49            Pen     1.723
 188      Subtask    108      ready     high   50            Key     0.001
 198      Subtask    108  sema wait     high   45            DmKK    0.059
 1A8      Subtask    108      ready      low   30            FSUI    0.165
 1C8      process          msg wait  med low    7    4096#   SEC0    0.121
 1E8      process          msg wait  med low    7    4096#   nbt0    0.088
 208      process          msg wait  med low    7    4096#   NBA0    0.089
 228      process          msg wait  med low    7       0#   note    0.031
 238      process          msg wait  med low    7    4096#   Min0    0.414
 258      process          msg wait  med low    7   28672    MIN0    0.264
 278      process          msg wait  med low    7    4096#   SET0    0.592
 2B8      process          msg wait  med low    7    4096#   HEL0    0.082
 2D8      process          msg wait  med low    7    4096#   INB0    0.086
 2F8      process          msg wait  med low    7    4096#   OBO0    0.087
 318      process          msg wait  med low    7    4096#   sta0    0.086
 338      process          msg wait  med low    7    4096#   CNC0    0.125
 358      process          msg wait  med low    7    4096#   VKE0    0.114
 378      process          msg wait  med low    7    4096#   MAS0    0.100
 398      process          msg wait  med low    7    4096#   ACC0    0.098
 3B8      process          msg wait  med low    7    4096#   CLO0    0.119
 3D8      process          msg wait  med low    7    4096#   BOO0    0.103
 3F8      process          msg wait  med low    7       0#   anim    0.007
 408      process          msg wait  med low    7       0#   hwle    0.012
 418      process          msg wait  med low    7       0#   hwge    0.010
 428      process          msg wait  med low    7  290816    ctsh    0.252
 448      process          msg wait  med low    7    4096#   flap    0.028
 468      process          msg wait  med low    7    4096#   SIO.   0.029
 488      process          msg wait  med low    7    4096#   HSLI    0.021
 4A8      process          msg wait  med low    7   20480    atp.   0.064
 4B8      Subtask    4A8      ready  med low    7            ATP     0.024
 4D8      process          msg wait  med low    7   28672    MAR0    0.496
 4E8      process          msg wait  med low    7   53248    BOO1    1.456
 4F8      process          msg wait  med low    7   32768    NBA1    1.204
 508      process          msg wait  med low    7   57344    nbt1    2.224
 548      process          msg wait  med low    7  135168    MAR2    2.357
 558      Subtask    548      ready  med low    7                    0.931
Memory allocated by the system:                    3977216
Memory allocated by the loader:                    2654208
```

There's task 558, a subtask our application created. You can then switch context to the task and get a stack trace and look at the instruction pointer.

```
>ctx 558
>u eip-20
 40774CDD 4D                   DEC     EBP
 40774CDE 7740                 JA      40774D20
 40774CE0 55                   PUSH    EBP
 40774CE1 8BEC                 MOV     EBP,ESP
 40774CE3 57                   PUSH    EDI
 40774CE4 56                   PUSH    ESI
 40774CE5 53                   PUSH    EBX
 40774CE6 8B6D08               MOV     EBP,[EBP+8]
 40774CE9 3E8B7D04             MOV     EDI,DS:[EBP+4]
 40774CED 66BACE03             MOV     DX,03CE
 40774CF1 66B80503             MOV     AX,0305
 40774CF5 66EF                 OUT     DX,AX
 40774CF7 66B808FF             MOV     AX,FF08
 40774CFB 66EF                 OUT     DX,AX
 40774CFD 668B451A             MOV     AX,[EBP+1A]
 40774D01 6683F801             CMP     AX,01
>
```

The Task List 12.7

It is fairly useful to understand what tasks appear in the task list. This section provides a road map to the significant portions of the task list.

The two most interesting columns are **type** and **name**. Following the header are a number of system tasks:

```
Task Id      type parent     state    class  pri  memused   name   time(s)
-------      ---- ------     -----    -----  ---  -------   ----   -------
  48      process            ready      low   50   172032   Idle   20.053
  68    Interrupt     48                                    Dbl     0.000
  78    Interrupt     48                                    TSS     0.000
  88*   Interrupt     48                                    Stak    0.000
  98      process         msg wait   system    9     8192   Scav    0.548
  A8      process         msg wait     high   44     4096   V86x   40.474
  B8    Interrupt     48                                    V86a    0.000
  C8      Subtask     48     ready med high    7            Page    0.847
  D8      Subtask     48  msg wait     high    3            Powr    0.145
  E8      Subtask     48     ready     high   45            DmIM    0.070
  F8      Subtask     48 sema wait  med low    7                    0.000
 108      process         msg wait  med low    7   835584   Syst    8.711
 118      Subtask    108     ready  med low    7            Timr    0.023
 128      Subtask    108 sema wait     high   46            DBm    21.608
 138      Subtask    108     ready     high   45            DBk     0.008
 148      Subtask    108  msg wait  med low    7                    0.359
 158      Subtask    108  msg wait  med low    7            IMgr    0.017
 168      Subtask    108 sema wait  med low    7            IDSP    0.771
 178      Subtask    108     ready     high   49            Pen     1.723
 188      Subtask    108     ready     high   50            Key     0.001
 198      Subtask    108 sema wait     high   45            DmKK    0.059
 1A8      Subtask    108     ready      low   30            FSUI    0.165
```

Idle Does nothing.

Dbl Interrupt task that handles double faults (that is, two simultaneous faults such as a page fault and a stack fault). This is Interrupt 8—Double Fault from the Intel book. It should never happen.

TSS Interrupt task that handles invalid task segment structures. This is Interrupt 10—Invalid TSS from the Intel book. It should never happen.

Stak Interrupt task that handles stack faults in ring 0. This is Interrupt 12—Stack Exception from the Intel book. Note that most code is in ring 3, and exceeding stack in ring 3 is handled in-line by growing the stack without the kernel doing anything special.

Scav Process that scavenges, activated by Class Manager.

V86x Virtual 8086 task. Calls DOS Int 13 functions in real mode, to avoid problems with trying to call these functions in protected mode. Note that tablet hardware doesn't have this task; it can use the MIL to talk to devices in protected mode.

V86a Handles interrupts in virtual 8086 mode.

Page Page daemon that writes out dirty pages. If you change data, or memory-map a file and change something in memory, the contents of memory are dirty and have to be written out (a basic virtual memory idea). This process does it periodically.

Powr Power management, turns off power, handles the physical power button.

DmIM "Dm" indicates a continuous Driver interface to the MIL. The last two letters represent the logical device (A is 0, so I is logical device 8) and the function (A is 0, so M is function 12). You can enter **`ldev`** in the mini-debugger to get a list of logical devices, then look in the DV*.H header to figure out its function.

Syst Loads DLL code that doesn't have a process. It is the owner for the resources allocated by these DLLs. Observe how much memory it uses (835584 bytes!). The DLLs mentioned in BOOT.DLC are all owned by this task.

Timr Keeps track of timer events. You can't send a message during a timer interrupt. Timer interrupts use a semaphore, which is checked by this subtask. When the semaphore is clear, the subtask starts timer processing and message sending.

DBm The main DB task.

DBk The DB keyboard reader.

IMgr The Input manager that grabs pen and keyboard events.

IDSP The Input dispatcher that dispatches input to windows.

Pen The pen subtask.

Key The keyboard subtask.

DmKK Another continuous Driver interface to the MIL.

FSUI Handles the file system user interface.

1C8	process		msg wait	med low	7	4096#	SEC0	0.121
1E8	process		msg wait	med low	7	4096#	nbt0	0.088
208	process		msg wait	med low	7	4096#	NBA0	0.089
228	process		msg wait	med low	7	0#	note	0.031
238	process		msg wait	med low	7	4096#	Min0	0.414
258	process		msg wait	med low	7	28672	MIN0	0.264
278	process		msg wait	med low	7	4096#	SET0	0.592
2B8	process		msg wait	med low	7	4096#	HEL0	0.082
2D8	process		msg wait	med low	7	4096#	INB0	0.086
2F8	process		msg wait	med low	7	4096#	OBO0	0.087
318	process		msg wait	med low	7	4096#	sta0	0.086
338	process		msg wait	med low	7	4096#	CNC0	0.125
358	process		msg wait	med low	7	4096#	VKE0	0.114
378	process		msg wait	med low	7	4096#	MAS0	0.100
398	process		msg wait	med low	7	4096#	ACC0	0.098
3B8	process		msg wait	med low	7	4096#	CLO0	0.119
3D8	process		msg wait	med low	7	4096#	BOO0	0.103
3F8	process		msg wait	med low	7	0#	anim	0.007
408	process		msg wait	med low	7	0#	hwle	0.012
418	process		msg wait	med low	7	0#	hwge	0.010
428	process		msg wait	med low	7	290816	ctsh	0.252
448	process		msg wait	med low	7	4096#	flap	0.028
468	process		msg wait	med low	7	4096#	SIO.	0.029
488	process		msg wait	med low	7	4096#	HSLI	0.021
4A8	process		msg wait	med low	7	20480	atp.	0.064
4B8	Subtask	4A8	ready	med low	7		ATP	0.024
4D8	process		msg wait	med low	7	28672	MAR0	0.496

Now we're getting into processes. The number on the end of the process name represents the **processCount** (the same thing that an application receives in **main**). Therefore, task names that end in 0 are the the application monitor objects created in **AppMonitorMain** when the applications were installed. This is also usually where the application creates its application class. There are a lot of these process 0 tasks; to save memory PenPoint compacts them. This is what the "#" means in the memory column.

SEC0 This is process 0 of the section application.

nbt0 Process 0 of the Notebook TOC app class.

anim This is a DLL that has a **DLLMain**. It's getting loaded after other DLLs because it isn't a system DLL in BOOT.DLC, it's an optional DLL. Any app that wants to use **clsAnimator** has to mention its DLL in the application's .DLC file, hence the other term for these DLLs, "distributed DLLs." PenPoint is smart enough to only load a DLL if it is not already loaded (assuming the loaded version matches).

hwle Another non-system, "distributed" DLL, the letter practice DLL.

hwge Ditto, the handwriting gesture practice DLL.

flap The low-level LocalTalk protocol.

SIO The serial I/O DLL.

atp AppleTalk Transport Protocol DLL.

ATP A subtask created by the ATP stack code.

```
 4E8      process         msg wait  med low   7    53248   BOO1    1.456
 4F8      process         msg wait  med low   7    32768   NBA1    1.204
 508      process         msg wait  med low   7    57344   nbt1    2.224
 548      process         msg wait  med low   7   135168   MAR2    2.357
 558      Subtask   548      ready  med low   7                    0.931
```

Now we're getting into non-zero application processes, which is where actual application objects are created (in **AppMain()**). In other words, these are processes associated with active documents.

BOO1 This is the bookshelf application, which is the first time a bookshelf document is activated. It's not surprising that this is the first document actually created; PenPoint creates the bookshelf from the line in ENVIRON.INI:

```
StartApp = \\boot\penpoint\boot\app\bookshelf
```

MAR2 An application processCount 2. In other words, this is the second time a document of this app has been activated. Note that it could be the same document both times—when you turn away from a document the app object may be terminated, which leads to the process being terminated.

(blank) A subtask created by the application task (548).

Part 3 / Tools

List of Figures

List of Tables

Chapter 13 / Introduction

Part 3: Tools describes the development tools provided with the PenPoint™ operating system SDK.

Some of the tools you use in DOS while developing your application; others are PenPoint tools that you use to refine or document your application.

Organization of Part 3 13.1

Chapter 14, DOS File System Utilities, describes DOS programs that you use to view or modify PenPoint-specific file system information.

Chapter 15, Other DOS Utilities, describes DOS programs that you use to compile method tables, compile and manipulate resource files, and eliminate unneeded clutter from Rich Text Format (RTF) Help notebook files.

Chapter 16, PenPoint Bitmap Editor, describes the PenPoint application that you can use to edit bitmaps, such as PenPoint application icons.

Chapter 17, S-Shot Screen Capture Utility, describes the PenPoint application that you can use to gather TIFF images of the screen. You can use these TIFF images in the documentation for your application (the PenPoint user's guides are full of S-Shot images).

Chapter 18, Font Editor, describes a font editor that you can use to create new fonts or edit existing fonts for the PenPoint operating system.

Chapter 14 / DOS File System Utilities

The PenPoint™ operating system's file system is designed to be layered on top of any existing file system. Currently, the only file system used by PenPoint is the DOS file system.

To support the PenPoint extensions to the DOS file system, PenPoint creates a PENPOINT.DIR file in each directory that contains PenPoint files. The PENPOINT.DIR file contains information for each of the PenPoint files in that directory.

In the course of application development, you need to create or modify PENPOINT.DIR entries for some of your DOS files. At other times, you may need to examine the contents of the PENPOINT.DIR file.

The PenPoint SDK includes a set of DOS programs that let you create and modify PENPOINT.DIR files from a PC. These programs are in \PENPOINT\SDK\UTIL\DOS, summarized in Table 14-1 and described in detail later in this chapter.

Table 14-1
DOS File System Utilities

Program Name	Purpose
STAMP.EXE	Adds special PenPoint information to a DOS file or directory.
GDIR.EXE	Lists the PenPoint names and file system attributes for all the files and directories in a DOS directory.
MAKLABEL.EXE	Computes the correct hexadecimal values for file system attribute labels.
PAPPEND.EXE	Appends directory entries from one PENPOINT.DIR file to another, without appending duplicate names.
PDEL.EXE	Deletes specific directory entries from PENPOINT.DIR files.
PSYNC.EXE	Scans the current directory and removes any entries from PENPOINT.DIR for which there are no corresponding files.

STAMP 14.1

The STAMP utility associates a long PenPoint operating system file name or attributes with a DOS file or directory by modifying or creating a PENPOINT.DIR file. The syntax for STAMP is:

STAMP [`path`] **/G** "*PenPoint Name*" [**/D** *dos-name*] [**/A** *attrlabel value* ...]

path An optional DOS path to the PenPoint directory where the file or directory exists. If not specified, STAMP tries to stamp the file in the current directory.

PenPoint Name The PenPoint node name. It must be in quotes.

dos-name The DOS file name of the file or directory.

attrlabel The 4-byte hexadecimal value of the file attribute label. The file attribute label type (FS_ATTR_LABEL) is defined in \PENPOINT\SDK\INC\FS.H.

value The attribute value. Its format depends on the label type. If the label type is **fsFixAttr** or **fsFix64Attr**, then the attribute requires a single 4-byte or 8-byte hexadecimal number. If the label type is **fsStrAttr**, the attribute requires a quoted string. If the label type is **fsVarAttr**, the attribute requires a set of two-digit hexadecimal numbers (for example, 3A CE 45); STAMP calculates the length of the variable attribute.

You can use the MAKLABEL utility to calculate a file system attribute label. *Part 6: File System*, in the *PenPoint Architectural Reference*, explains file system attributes in more detail.

The DOS name is optional once a PenPoint name has been associated. If the DOS name is not specified, STAMP assumes you want to add more attributes. It is OK to specify the DOS name as well as the PenPoint name; however, it is not correct to specify a DOS name that doesn't match the DOS name that the PenPoint name is associated with.

Example

14.1.1

This example associates the PenPoint name "Tic-Tac-Toe" with the directory \PENPOINT\APP\TTT\STATNRY\TTTSTAT1, and assigns the attribute `00800274` (with the value 1) to the file.

The attribute 00800274 causes the document to appear in the pop-up stationery menu.

```
C:> stamp \penpoint\app\ttt\statnry /G "Tic-Tac-Toe" /D tttstat1 /A 00800274 1
```

You can specify more than one attribute in a single command by adding as many **/A** parameters as you need. For example:

```
C:> stamp "Foo" /A 0000004F "Stuff" /A 0000005F "More Stuff"
```

GDIR

14.2

The GDIR utility lists PenPoint names and attributes in a specific directory.

GDIR [*path*]

path An optional DOS path to the directory to list. If not specified, GDIR lists names in the current directory.

If the specified directory contains a PENPOINT.DIR file, this utility lists the PenPoint name, number of attributes, and the attributes for each file and directory listed in PENPOINT.DIR.

MAKLABEL

14.3

The MAKLABEL utility constructs a hexadecimal attribute label from a class value and an index value.

MAKLABEL {**FIX** | **F64** | **STR** | **VAR**} *class index*

FIX | F64 | STR | VAR The first argument specifies the type of file system attribute, corresponding to **fsFixAttr**, **fsFix64Attr**, **fsStrAttr**, and **fsVarAttr** in \PENPOINT\SDK\INC\FS.H.

class The administrated number of the attribute's class, in decimal.

index The attribute's index, in decimal.

MAKLABEL generates the hexadecimal file system attribute labels required by STAMP. It has similar parameters as the **FSMakeAttr** macro in \PENPOINT\SDK\INC\FS.H which are used to define file system attributes in header files.

Example 14.3.1

\PENPOINT\SDK\INC\AUXNBMGR.H defines the file system attribute, which indicates that stationery should be on the pop-up stationery menu:

```
// Should a piece of stationery be on the stationery menu?
#define anmAttrStationeryMenu       FSMakeFix32Attr(clsAuxNotebookMgr, 1)
typedef enum ANM_ATTR_STATIONERY_MENU {
    anmNotOnMenu            = 0,      // Same as no attribute
    anmOnMenu               = 1
} ANM_ATTR_STATIONERY_MENU;
```

\PENPOINT\SDK\INC\UID.H gives the administered number for **clsAuxNotebookMgr**:

```
#define clsAuxNotebookMgr       MakeWKN(314,1,wknGlobal)
```

So, the command to print out the hexadecimal value of **anmAttrStationeryMenu** is:

```
C:> \penpoint\sdk\util\dos\maklabel fix 314 1
The attribute label = 00800274
```

This gives the correct value for the **attrlabel** parameter to STAMP.

PAPPEND 14.4

The PAPPEND utility appends directory entries from one PENPOINT.DIR file to another. Before appending an entry, PAPPEND makes sure that the entry doesn't already exist in the target file. If the entry exists already, PAPPEND does not append that entry. PAPPEND also eliminates any empty directory entries from the source while appending.

The syntax for PAPPEND is similar to the standard C function `strcat`:

PAPPEND *target-file source-file* [**/G** "*PenPoint-name*"] [**/V**]

target-file The file to which you are appending a directory entry.

source-file The file from which the entries are being copied.

/G "**PenPoint-name**" Specifies that PAPPEND is to append only the entry with this name to the target file.

/V Directs PAPPEND to tell you what it has done (verbose mode).

Example 14.4.1

When you copy an application directory to \PENPOINT\APP, you need to add the entry for that application directory to PENPOINT.DIR. The easiest way to do this is to append the entry from the application directory's source disk.

```
B:\PENPOINT\APP> pappend penpoint.dir e:penpoint.dir /g "DrawingPaper" /v  Appended
"DrawingPaper"
```

PDEL 14.5

PDEL deletes a specific entry from the PENPOINT.DIR file in your current directory.

The syntax for PDEL is:

PDEL "*PenPoint-name*" [/B] [/D "*directory*"] [/V]

PenPoint-name Specifies the PenPoint operating system name of the file to delete from PENPOINT.DIR.

/B Directs PDEL to create a backup file named PENPOINT.BAK.

/D "***directory***" Directs PDEL to find PENPOINT.DIR in the specified directory.

/V Directs PDEL to work in verbose mode.

PSYNC 14.6

PSYNC examines the PENPOINT.DIR file in your current directory and compares the directory entries to the actual files in that directory. When it finds an entry for a file that doesn't exist, PSYNC removes the entry from PENPOINT.DIR.

The syntax for PSYNC is:

PSYNC [/B] [/D "*directory*"] [/P] [/V]

/B Directs PSYNC to create a backup file named PENPOINT.BAK.

/D Directs PSYNC to find PENPOINT.DIR in the specified directory.

/P Directs PSYNC to prompt you before removing a directory entry.

/V Directs PSYNC to work in verbose mode.

Chapter 15 / Other DOS Utilities

In addition to the file system utilities, the PenPoint™ operating system provides a number of other DOS utilities that are used in developing applications and running PenPoint.

GO Batch File 15.1

\PENPOINT\SDK\UTIL\DOS\GO.BAT is a batch file that starts up PenPoint on a PC. You can modify this batch file to ensure that PenPoint removes TSRs (terminate and stay resident) before running PenPoint, or to have PenPoint check hard disk consistency after running PenPoint.

GO.BAT is explained in more detail in Chapter 3, Running PenPoint on a PC.

PenPoint Method Table Compiler 15.2

\PENPOINT\SDK\UTIL\CLSMGR\MT.EXE is the PenPoint method table compiler. It compiles a method table source file into an object file that you link with your class's code. Its use is explained in the *PenPoint Application Writing Guide* and in *Part 1: Class Manager* of the *PenPoint Architectural Reference.*

Resource Utilities 15.3

You can create PenPoint text resource files two ways: programmatically (from within a PenPoint application) or by hand. Usually when you create something like a quick-help resource, you build it by hand, then compile using the PenPoint resource compiler (RC). RC enables you to create or append to PenPoint resource files.

The executable for RC is in \PENPOINT\SDK\UTIL\RC.EXE.

To append resources from one resource file to another, use RESAPPND. RESAPPND also compacts resource files by removing deleted or duplicate resources.

For example, MAKEFILE for the Tic-Tac-Toe sample application uses RESAPPND to append all of its separate resource files to its APP.RES resource file, thereby creating its application resource file.

RESAPPND is in \PENPOINT\SDK\UTIL\RESAPPND.EXE

To view the contents of a resource file, you can use RESDUMP. If you want to decode the state of a PenPoint document, you can use PenPoint to copy the document from the TOC to a disk browser, then under DOS use RESDUMP to dump the contents of the document's DOC.RES file.

RESDUMP is in \PENPOINT\SDK\UTIL\RESDUMP.EXE.

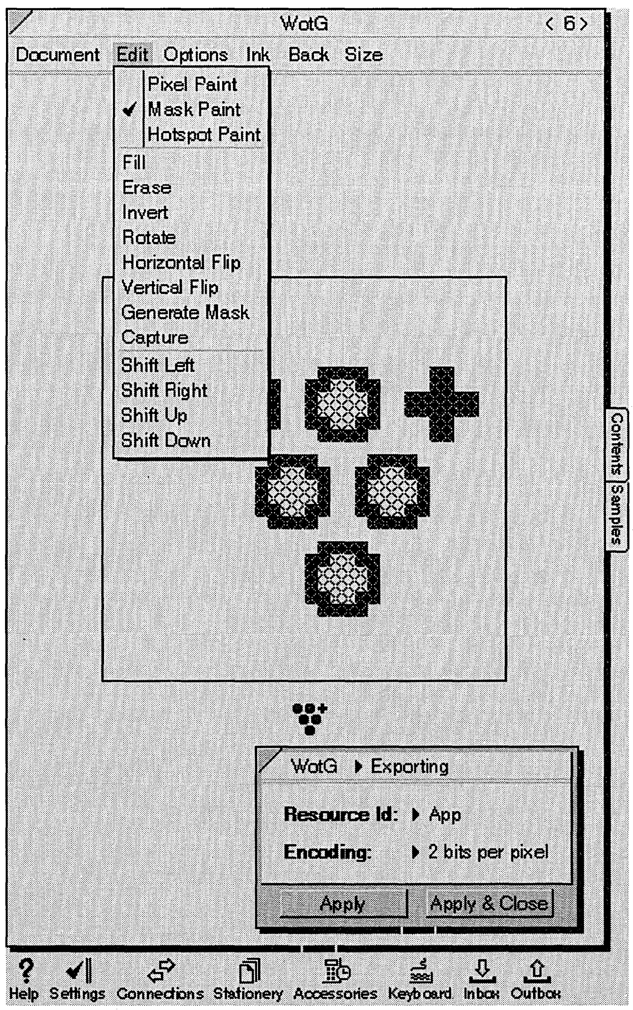
For more information on resource files and resource compiling, see *Part 11: Resources* in the *PenPoint Architectural Reference.*

Chapter 16 / PenPoint Bitmap Editor

The user interface to the PenPoint™ operating system uses icons to identify items in the Bookshelf, Notebook, cork margin, accessories and other notebooks.

You can use the bitmap editor to create **icons** and small **sampled images** (images made of pixels). The bitmap editor has a fairly comprehensive set of tools for basic editing of small **pixelmaps**: foreground and background colors, image shifting, capture from the screen, and so on.

Figure 16-1
PenPoint Bitmap Editor



Most applications use two icons: a large one (32 by 32) and a small one (16 by 16). By default, the small icon is used for items on the Bookshelf and in Accessories (draw a check over any icon on the Bookshelf to change style). The small icon is also displayed next to documents in a table of contents or other browser display.

When an application has bitmaps for these two icons in its resource file, the PenPoint operating system displays them in the appropriate situations. Otherwise, it uses default icons from the system resource file.

Elements of a Bitmap 16.1

You can use the bitmap editor to access and change three basic bitmap elements:

- The array of pixels that make up the bitmap's image.
- The array of bits that indicates which pixels will be visible when the bitmap is displayed. This array is called the **mask**. Only if a bit in the mask is on will its corresponding pixel be visible when the bitmap is displayed.
- The hot spot, which locates the "origin" of the bitmap. This, for example, is the pixel in the bitmap that will be drawn underneath the pen when the bitmap is being used as a cursor.

The bitmap editor has three modes, one for editing each of the three elements described above: Pixel Paint, Mask Paint, and Hotspot Paint.

In Pixel Paint mode, you can edit the bitmap's pixels. When you drag the pen, the pixels that the pen touches change, depending on the color of the pixel on which you started.

- If you started on a pixel that is not in the current ink color (any of the other three colors), the pen will draw that color.
- If you started on a pixel that is the current ink color, the pen will draw the current background color.

In Mask Paint and Hotspot Paint mode, only the mask or the hot spot are affected by dragging the pen in the bitmap editor.

Importing a Bitmap 16.2

To import a bitmap from an existing document, use the Connections notebook to find the application's resource file (such as \PENPOINT\APP\Some App\APP.RES), tap-press to copy it, then drag the copy into the table of contents of the Notebook. A menu of possible import types will appear: choose Bitmap Editor; for possible bitmap types, choose either App or Small App.

To find a file in the \PENPOINT\APP directory, you must have the B 800 debug flag set.

A new bitmap editor document will appear containing the bitmap. The document remembers where the bitmap resource came from, and will replace it in the resource file.

For more information on importing documents, see PenPoint's end-user manual, *Using PenPoint.*

Exporting a Bitmap 16.3

As mentioned above, the PenPoint operating system looks in an application's resource file for the application's bitmaps. The application's resource file is the file APP.RES in the application's directory. For example, the Tic-Tac-Toe program's application resources, including bitmaps, are in \PENPOINT\APP\TTT\APP.RES. Some programs create a resource file in advance; others only have a resource file after installation under PenPoint.

Once you have created a bitmap for an application, you must export that bitmap to the application's resource file so that PenPoint can access it.

To export a bitmap, tap on the Exporting menu button under Options. The Resource ID field on the Exporting card is a pop-up choice, which allows you to specify which of the two application icon resources use this bitmap (large or small).

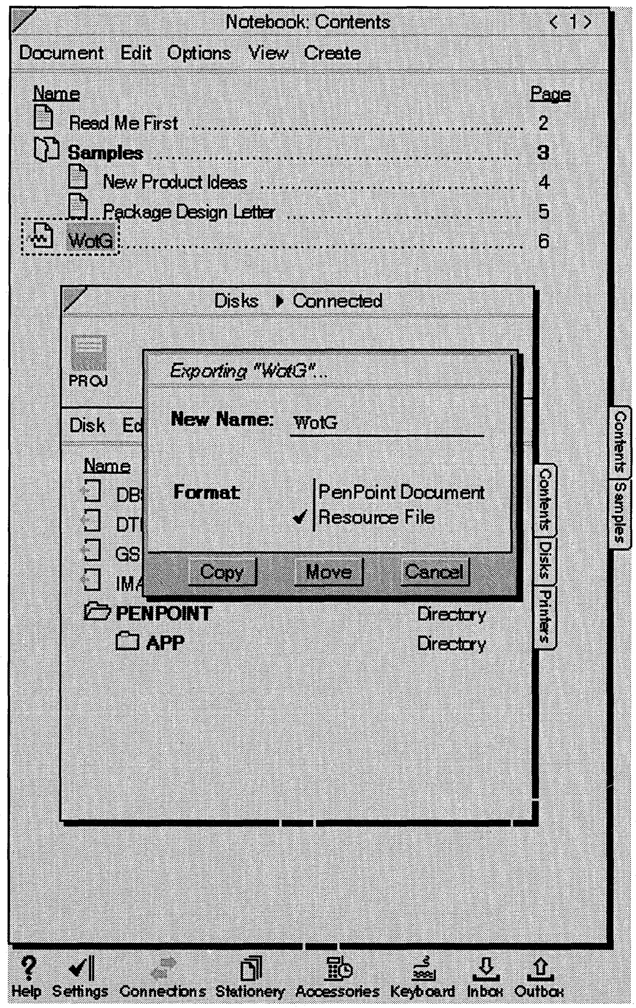
The only two icon resources used for application's bitmaps are the first two: App for the 32 by 32 icon and Small App for the 16 by 16 icon. Depending on the user's preference, PenPoint will use one or the other of these icon resources when displaying the application's icon. The other (Custom) resource ID is used by other parts of PenPoint and is usually not relevant for applications.

Once you have chosen which resource to use, apply the change and close the option sheet. You then export the bitmap by turning to the table of contents and moving or copying the bitmap document that you just modified to a disk file. PenPoint tells you that it can export the file and prompts you for the file format. Tap on Resource File.

You can export a bitmap to an existing resource file, or create a new one; if the existing resource file already contains a bitmap, PenPoint asks you if you want to replace the existing bitmap. You can also use the RESAPPND utility to append bitmap resources.

For more information on exporting documents, see *Using PenPoint*, PenPoint's user manual.

Figure 16-2
Exporting a Bitmap



Exporting to Home 16.4

When you import a bitmap from an existing resource file, the bitmap editor remembers the path to the resource file from which it came (including the volume name). When you are ready to write the bitmap to its resource file, you can tap on Export to Home in the Document menu.

You can see the path to the bitmap's home in the Exporting option card; however, you cannot modify the path.

Bitmap Editor User Interface 16.5

The application menu for the bitmap editor contains menus for Document, Edit, Options, Ink, Back, and Size. This section describes the Edit, Options, Ink, Back, and Size menus.

There is no undo capability in the bitmap editor. However, if you want to save your work, you can always tap Checkpoint in the Edit menu to save the document in its current state. If you make an unintended modification, you can then use Revert to return the bitmap to its state when you last used Checkpoint.

The Document Menu 16.5.1

The Document menu contains standard document menu buttons.

If you imported the current bitmap from a resource file, **Export to Home** also appears in the document menu. This non-standard item allows you to export the bitmap back to the resource file from which it came.

The Edit Menu 16.5.2

You choose an editing mode from the first three entries in the Edit menu:

Pixel Paint Displays only the bitmap's pixels so you can edit them.

Mask Paint Displays the bitmap's pixels and its mask. You can only edit the mask (indicated by an "X" over masked pixels). The masked pixels are the only ones that will be painted when the bitmap is displayed.

Hotspot Paint Displays the bitmap, but only lets you set the hot spot for the bitmap. The hot spot is indicated by an "X" in this mode. Hotspot mode is only of interest for bitmaps that will be used as cursors. (There is currently no way for an application to install a custom cursor.)

The other entries in the Edit menu are:

Fill Fills the entire bitmap area with black in Pixel Paint and Hotspot Paint. In Mask Paint, fill turns the entire mask on.

Erase Fills the entire bitmap area with the current background color in Pixel Paint and Hotspot Paint. In Mask Paint, erase turns the entire mask off.

Invert Inverts the colors in the bitmap in all modes. The bitmap has four possible colors: black, dark gray, light gray, and white. In inversion, black and white are switched and dark gray and light gray are switched.

Rotate Rotates the bitmap by 90 degrees clockwise.

Horizontal Flip Flips the bitmap around the y-axis.

Vertical Flip Flips the bitmap around the x-axis.

Generate Mask Turns the mask on for each pixel that is not the background color. The mask includes any closed areas in the bitmap (those that are surrounded by pixels that are not the background color).

Capture Displays a square on the screen that tracks the pen. When the pen is lifted, the area under the square is copied into the bitmap editor.

Shift Left, **Shift Right**, **Shift Up**, and **Shift Down** Shift the bitmap by one pixel (bit) in the indicated direction. Pixels that are shifted out of the bitmap on one side reappear on the other side (rotational shift).

The Options Menu 16.5.3

The Options menu provides standard access to the Controls, Access, and Comments option cards. The two interesting option cards added by the bitmap editor are:

Exporting Allows you to specify the Resource ID for an application icon.

Custom Resource ID Allows you to specify the Resource ID for a custom icon.

The Exporting Option Card 16.5.3.1

You use the About Exporting option card to specify how the bitmap editor is to export a bitmap.

Resource ID Allows you to specify the Resource ID for the bitmap. This is a pop-up choice that provides you with two predefined PenPoint Resource IDs (App and Small App) and a Custom Resource ID. Most applications will only use App and Small App. You should set the Resource ID to App when editing a 32 by 32 bitmap, and to Small App when editing a 16 by 16 bitmap.

Encoding Allows you to specify how many bits per pixel to use when exporting the bitmap. To save space, this should be as small as possible. For bitmaps that use 4 gray values, the encoding should be 2 bits per pixel.

The Custom Resource ID Card 16.5.3.2

If you are creating a bitmap that needs a Resource ID other than the two offered in the Resource ID pop-up choice, you can:

- Choose Custom on the Exporting option card.
- Tap on Apply.
- Turn to the Custom Resource ID option card.

The Custom Resource ID card allows you to specify a Resource ID for the bitmap.

Applications can display bitmaps with Custom Resource IDs by using **clsBitmap** or **clsIcon** to read in the Resource ID. For more details, see parts 3 and 4 in the *PenPoint Architectural Reference* and the header files for **clsBitmap** and **clsIcon**.

The three fields on the card specify the Resource ID (as defined in \PENPOINT\SDK\INC\RESFILE.H):

- The Resource Class field contains the administered number for the class that defines this Resource ID.
- The Resource Scope field specifies whether the Resource ID is global, process global, or private.
- The ID field contains an ID for this particular bitmap resource.

In your code, you'll probably define the same Resource ID using **MakeWknResId**(*tag*, *id*). *tag* identifies the class field and the scope, which is usually a class UID; *id* is a number between 0 and 255 that identifies the resource for this class.

For example, the Resource ID for the "SmallApp" resource used by PenPoint consists of:

- The resource class for **clsApp**, which is 13.
- The Resource ID, which is global.
- The ID value, which is 1.

To read this resource from a resource file:

```
#define clsApp MakeGlobalWKN(13, wknGlobal)

    RES_FILE              file;

    RES_READ_OBJECT     read;
    OBJECT              bitmap;
    ObjectCall(msgNewDefaults, clsObject, &read.new);
    read.mode           = resReadObjectMany;
    read.resId = MakeWknResId(clsApp, 1);
    ObjectCall(msgResReadObject, file, &read);
    bitmap              = read.object.uid;
```

Once you have read in the resource, you can use **clsIcon** or **clsBitmap** to display the bitmap.

The Ink Menu 16.5.4

The Ink menu allows you to set the ink color for painting operations when in Pixel Paint or Hotspot Paint mode.

This setting is ignored in Mask Paint mode.

The Back Menu 16.5.5

The Back menu allows you to set the background color for painting operations when in Pixel Paint or Hotspot Paint mode.

This setting is ignored in Mask Paint mode.

The Size Menu 16.5.6

You use the Size menu to change the bitmap's size. The possible sizes are: 8 by 8, 16 by 16, 24 by 16, 32 by 32, 40 by 32, and 48 by 32.

Caution When you change size, the bitmap clears.

- Use 32 by 32 size for large icons such as the App resource.
- Use 16 by 16 size for small icons such as the Small App resource.
- The 8 by 8 size is used within PenPoint for some cursors.

Chapter 17 / S-Shot Screen Capture Utility

S-Shot is a PenPoint™ operating system application that allows you to capture an image of the PenPoint screen. It saves the image to disk as an uncompressed gray-scale TIFF file, which you can then import into a variety of paint or drawing programs on a PC.

Using S-Shot 17.1

S-Shot essentially provides a user interface to ImagePoint's **msgDcScreenShot** message. This captures part of the screen in a TIFF file. To capture any portion of the screen, S-Shot disables window clipping.

One full-screen snapshot takes about 130 kilobytes.

If you are taking screen shots on a PC, you can store screen shots on your PC's hard disk. On a PenPoint computer, you'll need to make sure there is room on your hard disk, or some removable media, to store the screen shots.

Installing S-Shot 17.1.1

S-Shot is part of the PenPoint SDK; its files are in \PENPOINT\SDK\APP\SSHOT. You can include S-Shot in PenPoint running on a PC by adding a line to \PENPOINT\BOOT\APP.INI.

Using S-Shot 17.1.2

S-Shot is an accessory. To use it, open the Accessories notebook and tap on S-Shot.

The S-Shot window allows you to specify:

- The area that will be included in the screen shot.
- The delay before the screen shot is taken.
- The path to the file where S-Shot will store its TIFF image.

Specifying an Area 17.1.2.1

S-Shot can create an image from the entire screen (tap on the Full Screen button) or a specific area of the screen. To specify an area of the screen, you can do one of two things:

- Specify the size x-y and origin x-y location by entering specific values.
- Tap on the Set Area... button, which allows you to select the area visually.

If you tap the Set Area... button, S-Shot clears its window and displays a rectangular box in the center of the screen. This box delineates the area that S-Shot will capture.

You can expand or contract the box by placing the tip of the pen on one of the lines and dragging the line up, down, left, or right.

You can move the box by placing the tip of the pen in the center of the box and dragging the box wherever you want it to go.

Specifying a Delay 17.1.2.2

Obviously, you want the screen to look a certain way when you take the snapshot. Before capturing a screen, you should get its state close to the state you want, but there are two obstacles:

- The S-Shot window is in the way. When you take the snapshot, S-Shot takes down its window, but it takes time for the applications underneath the S-Shot window to repaint. This may take several seconds.
- You want to take a snapshot of a transient visual, such as ink on the screen, or quick help, or a highlighted menu button.

To take a broader range of snapshots, you can set a delay before S-Shot takes its snapshot. The default is five seconds, which should be enough time for the screen to repaint and for you to get the screen set up as you desire.

If you need more time, change the value.

Specifying a File Name 17.1.2.3

The default file name is \\BOOT\FILEA.TIF. This is appropriate for taking a screen-shot on a PC running PenPoint, because \\BOOT will be the hard disk containing the \PENPOINT hierarchy. However, on a PenPoint computer, you *must* change the file name to another name.

1. Tap on the File Name field to bring up a editing pad. The name you use *must* consist of two back-slashes, a volume name, a back-slash, and a file name.

   The volume name should be the DOS volume label of the disk on which you want to create the TIFF file. Use the DOS LABEL command to label a floppy disk or check its label.

   The file name could be any valid PenPoint file name. However, since the main use of S-Shot is to create TIFF files for editing on other computers, it's a good idea to adhere to the DOS file-naming conventions (no file name may have more that 8 characters). Leave the extension as .TIF. The writing pad restricts characters to uppercase.
2. Tap **OK** to apply your changes and dismiss the editing pad.

Do It 17.1.3

When you are ready to take the snapshot, tap Take Snapshot.

If the volume in S-Shot's file name is not connected, PenPoint displays a note prompting you to insert that volume.

The S-Shot window disappears from the screen, causing windows underneath it to repaint.

PenPoint will beep every second as it counts down the delay, then will beep a different tone. At this point it captures the screen to a TIFF file on disk. A full-screen snapshot takes about a minute to write to disk. During this time PenPoint will be frozen.

When the snapshot is complete, the S-Shot window reappears.

You will notice that S-Shot increments the snap file name in its window, so that if the file name was \\BOOT\FILEA.TIF before, it becomes \\BOOT\FILEB.TIF. If this is a reasonable name, you can keep it and take the next snapshot.

Hints on Using S-Shot 17.2

Taking Snapshots of Gestures 17.2.1

Set the delay high enough to give you time to do what you want. When you draw the gesture, don't lift the pen.

Impossible Snapshots 17.2.1.1

You can't take a picture of the screen during a page turn, or with the busy clock on the screen.

It's difficult to take a snapshot of disk operations such as the Disk Manager, import/export, or installation, because S-Shot prompts you to install its own volume, which changes the status of connected volumes.

Full-Screen Snapshots 17.2.2

The GO Computer screen is 640 pixels high by 400 wide.

Using S-Shot Files 17.2.3

On an Apple Macintosh Computer 17.2.3.1

You can use Apple File Exchange on a Superdrive-equipped Apple Macintosh computer to read from DOS-formatted 3-1/2 inch floppies, or use some cross-platform netword, such as SITKA's TOPS or Novell's Netware to transfer files.

To read a TIFF file into Macintosh programs such as Aldus FreeHand and Studio/8, you need to change the type of the file to TIFF using a Macintosh utility such as ResEdit, DiskTop, or DiskTools (in advanced mode).

IBM-Compatible PC 17.2.3.2

The files that S-Shot creates can be imported directly into a number of programs that run under MS-DOS, such as Microsoft Word, WordPerfect, Micrografx Designer, and so on.

Just insert the disk into your PC and choose your program's **Import TIFF** option.

TIFF Problems 17.2.3.3

There are many TIFF file variants, and most programs don't read in all flavors. In S-Shot, ImagePoint™ (the PenPoint operating system's imaging software) creates a "standard" uncompressed 4-bit gray-scale TIFF image. If your chosen PC application cannot read it, contact your vendor for assistance.

Bugs 17.2.4

S-Shot doesn't validate the numbers you enter for coordinates, or the name you give for the file.

S-Shot doesn't tell you when your disk is full, it just creates a small file.

Chapter 18 / Font Editor

The PenPoint font editor, FEDIT, enables you to create and modify outline 8-bit font files for the PenPoint™ operating system.

FEDIT runs under Microsoft Windows, version 3.0 or higher.

With FEDIT you can:

- Create and manipulate outline shapes.
- Create and manipulate hints.
- Integrate bitmaps as a part of a font.
- Read and write fonts in PenPoint font format and in other font formats.

Introduction and Concepts 18.1

The 8-bit characters that you create with FEDIT are identified using the AFII (American Society for Font Information Interchange) numbering scheme. While PenPoint 1.0 uses 8-bit characters, future releases of PenPoint will use 16-bit characters. The scheme for identifying 16-bit characters is not available at this time. Be aware that 8-bit characters created in FEDIT will be incompatible with the future 16-bit characters. In all likelihood, GO will not extend FEDIT to support 16-bit characters.

If you use a font that you created with FEDIT for special decorations in your user interface, we recommend that you use icons (which you create in the bitmap editor).

Getting Started 18.1.1

Run the program \PENPOINT\SDK\UTIL\DOS\FEDIT.EXE from within Windows. A blank window will appear. To edit an existing PenPoint font file:

1. Choose Open under the File menu. A standard Windows file-open dialog will appear.
2. Choose a file to open. A PenPoint font file should have the extension .PCK. Depending on the complexity of the font, it will take a moment for FEDIT to initialize the opened font database.
3. Choose the Choose command under the Character menu. A character selection dialog appears. It lists all the characters in the font. You can scroll the character list with the scroll bar below it.
4. Click on the character you wish to edit and click on the OK button. The main window will now repaint and you're ready to edit the shape of this character.

To create a new font file:

1. Choose New under the File menu.
2. Choose New under the Character menu. A dialog box appears. Enter the AFII number and the setwidth of the new character. The AFII number identifies a specific character; the setwidth specifies the amount of space that follows a character.
3. The main window repaints with an empty edit window. You're ready to add shapes to the new character.

File Formats 18.1.2

FEDIT supports several kinds of font files: the Nimbus-Q format, GO's proprietary packed format, and a limited support of Adobe's Type I font file. You may specify the input or output format to one of these three with the Options menu.

The extension of a Nimbus-Q file is .FNT; GO's file extension is .PCK; Adobe's file extension for Type I font is PFB.

You can easily convert the formats. For example, to convert a Nimbus-Q file to a GO packed file, use the following steps:

1. Choose Nimbus-Q /Input under the Options menu.
2. Open the Nimbus-Q file with the Open command under the File menu.
3. Choose GO packed Output under the Options menu.
4. Choose Save As under the File menu. You'll be prompted for an output file name. The saved file now has the GO packed format.

You can convert a GO packed format to the Nimbus-Q format by slightly reversing the procedure. Note, however, that the Nimbus-Q format does not have provisions for saving any bitmap rendering of the font. Thus, if a GO font contains bitmaps and you save it as a Nimbus-Q font, the bitmaps will be discarded. The same is true of converting a GO packed format to the Adobe Type I font format.

Technical Notes on Character Composition 18.1.3

A character is composed of the following elements:

- Its AFII number, which is its identity in the font. FEDIT takes a 16-bit signed representation for this field for ease of use but the sign has no significance.
- A setwidth, which specifies the amount of space to be added after the character has been rendered.
- One or more closed shapes, which define the appearance and the placement of the character. For simplicity, a closed shape will simply be referred to as a shape. Each shape is composed of connected straight lines or Bezier curves. Each line or curve in a shape is called a segment. The lines and curves that

define a shape have direction: clockwise or counter-clockwise. Direction is important when filling and merging.

Font Units

18.1.3.1

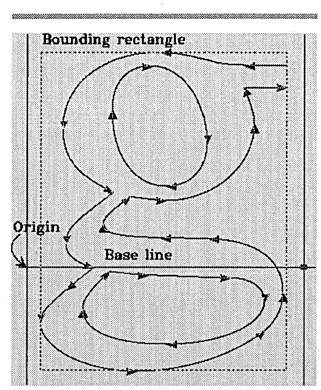
The setwidth and the control points that define the segments are in the unit of the integer Cartesian grid of [-2048, 2047] x [-2048, 2047]. They are called font units. To render a font with a particular point size, rotation, or italicization, the setwidth and the control points are transformed by a transformation matrix that contains the scaling, rotation, and so on, with 1000 as the standard scaling factor.

For example, to render a 12-point font on a 300 dpi device, the scaling factor will be (12 x 300) / (72 x 1000). Although you'd never need to compute these numbers for FEDIT, you need a rough grasp of sizes and proportions to design a good font. For example, a font whose shapes are all inside the square grid of [300, 300] x [300, 300] is undesirable because even a large point size will produce only a very small font. Most popular fonts, such as Times Roman and Helvetica, have characters sized around 1000 x 1000.

Character Placement

18.1.3.2

The set of shapes that make up a character implicitly defines a rectangle that minimally bounds all the shapes in that character. This is called, not surprisingly, the **bounding rectangle** of the character.

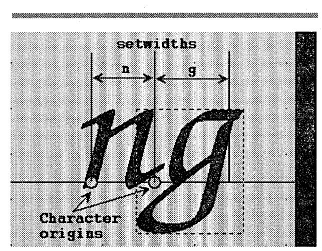


How does the bounding rectangle affect the placement of characters next to each other in a string? For simplicity, let us assume that we are rendering a string on a device where one font unit is one device pixel; that is, the transformation matrix is the identity, along the x-axis.

First note the position of the bounding rectangle relative to the x-axis, or the base line. The base line is the common reference line that allows the characters in a string to line up. Thus the base of most upper case letters should be directly above or very slightly below the base line while the tails of characters such as g, p, or q will extend far below the base line.

Recall that the setwidth of character Xis the value added to the x-coordinate of the cursor each time the character X is drawn. (The setwidth is quite independent on the character's bounding rectangle.) The new x-position of the cursor is where we'd place the origin (0, 0) of the next character.

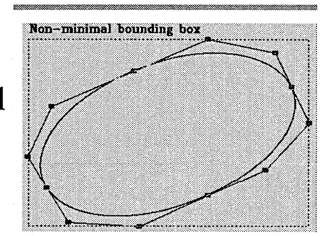
The bounding rectangle of the next character is thus important in determining where it is placed after the setwidth of the previous character has been added to the cursor. If, for example, the origin of the character lies inside its bounding box, its left-most edge will be placed to the left of the new cursor position. It is also possible for the setwidth to be inside the character's bounding rectangle so that the bounding rectangle of the next character to be drawn in the string can overlap the current character. This is useful for a "pseudo-kerning" effect as illustrated on the right.



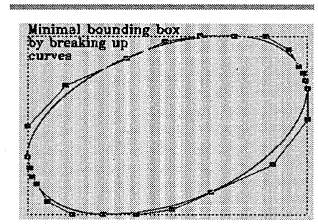
You may examine these character metrics in a dialog box with the Info command under the Character menu. In addition, the Info command lists the number of lines and curves in the character. You can also alter the AFII number of the character in this dialog box.

Control Point Placement 18.1.3.3

A **Bezier curve** is composed of four control points, two of which are the end points of the curve, while the other two lie outside of the curve (called the control points). The bounding rectangle of a character is computed based on the coordinates of the control points. The bounding rectangle is *not* based on the pixels that would be drawn if we render the character.



It is possible (but highly discouraged) for a character to have a bounding rectangle that minimally bounds the control points, but not be minimal with respect to the actual pixels that would be drawn if we render the character. The font subsystem in PenPoint computes character metrics based on the assumption that the bounding rectangle of a character bounds not only control points, but rendered pixels as well.



It is always possible to break up a Bezier curve to satisfy this requirement using the mitosis operator described below.

Adding and Deleting a Character 18.1.4

To add a character, choose the New command under the Character menu and specify the new character's AFII number and setwidth in the dialog. Each character in a font must have a unique AFII number. FEDIT will not allow you to assign an AFII that is already used to a new character in the font. Each font file can hold up to a maximum of 512 characters.

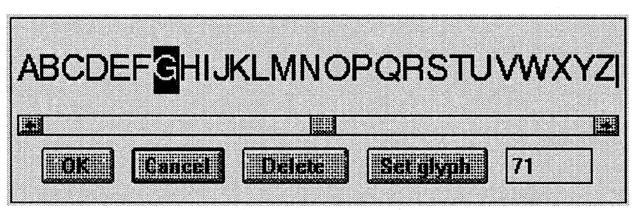
To delete a character, choose the Choose command under the Character menu to invoke the character list dialog. Click on the character you wish to delete, then click on the Delete button.

Caution *Character deletion cannot be undone.*

Editing Character Shapes 18.2

To edit character shapes, you first choose a character from the character selection window. To bring up the character selection window, click on the Choose command from the Character menu. The window shown in Figure 18-1 appears.

Figure 18-1
The Character Selection Window



- To choose a character for shape or hint editing, click on the character, then click on the OK button (or just double-click on the character).
- To delete a character, click on the character and click on the Delete button.
- To change the AFII number of a character, click on the character. The current AFII number appears on the leftmost box. Type in the new AFII number and click on the Set Glyph button.

Caution None of the operations executed from this window are undoable. Think twice before you delete a character.

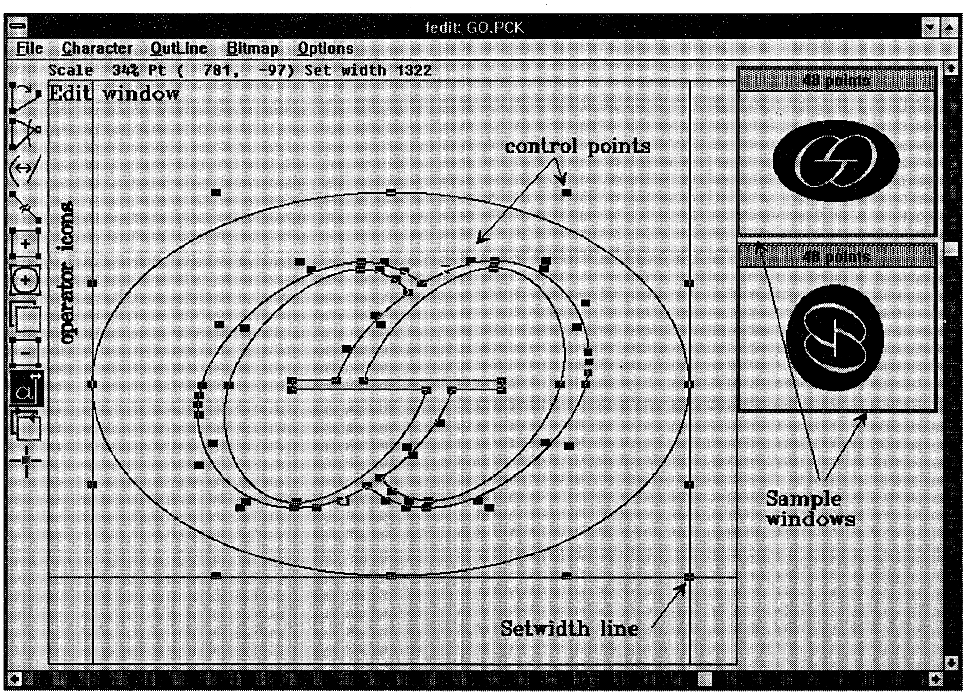
The Outline Editing Window

18.2.1

To edit the shapes in a character, click on a character in the character selection window, then click on the Shape command from the Outline menu. The outline editing window will appear. Figure 18-2 shows the outline editing window and identifies some of the terms in the window.

- The edit window is where you manipulate the shapes that make up the character.
- The sample windows show two samples of the rendering of the current shape: one portrait, one landscape (rotated 90 degrees). When you drag the borders of these windows, the point size of the samples change; this allows you to examine the character at different point sizes.
- When you click in either of the sample windows, the sample windows update themselves to reflect the shape of the character in the edit window. The sample windows do not redraw themselves each time you alter the shapes in the edit window.

Figure 18-2
The Outline Editing Window



- To zoom into a portion of a character for fine editing, click and drag in the portrait sample window; as you drag, a zoom rectangle will appear. Where you first click is the center of the rectangle. When you release the mouse button, the edit window will show that portion of the character you have just selected. The scale status field shows you the scale factor relative to font units at which the character (or part of it) is rendered in the edit window. Thus, 100% means that one font unit equals one device unit on the screen.
- When you drag out a zoom rectangle, holding down the left mouse button adjusts the size of the zoom rectangle. To move the zoom rectangle, hold down both the left and right buttons.

The coordinate values at the top of the edit window show the position of the cursor in font units. Note that the cursor moves in the screen device's resolution, which is usually at a lower resolution than the font units. FEDIT can only scale the font coordinates from device coordinates. These numbers are therefore not exact.

- You can pan around the character with the horizontal and vertical scroll bars.
- The setwidth line (with a small control box at the base line) shows the current setwidth of the character.
- The numerous small, filled boxes in the edit window are the control points for the shapes.
- The column of icons at the left side of the window are the operator icons. You select one of these icons to invoke an editing function. (The rest of this chapter explains the action of these icons.)
- All of the shape operator icons will cause part or all of the character window to redraw. Bezier curves are rendered in the edit window using line segment approximation. The BezierResolution command in the Options menu controls how fine the line segments should be. Low resolution is quick and adequate for highly interactive editing. If you want to view the curves in high resolution, which will be slow in rendering, choose the high option.

View Preference 18.2.2

To toggle the information displayed in the edit window, you can click on View Preferences in the Option menu. In the View Preferences dialog:

Top/Bottom Draws the horizontal lines representing the maximum ascender and minimum descender for the entire font.

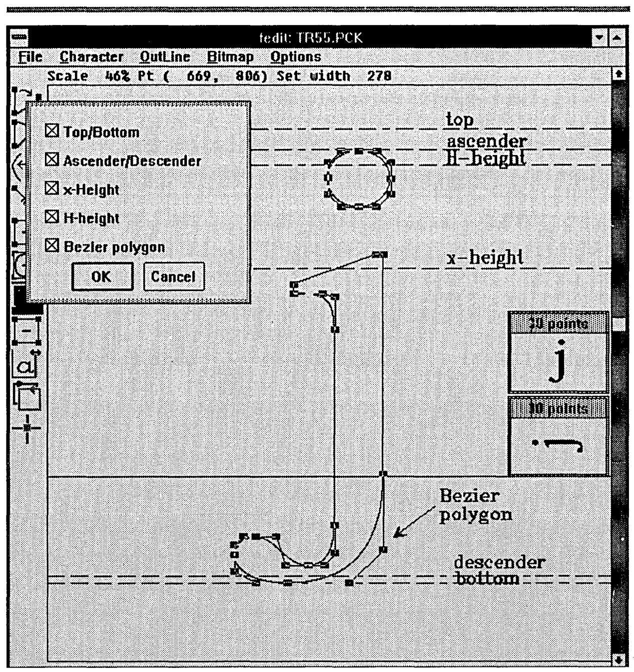
Ascender/Descender Draws the horizontal lines representing the nominal ascender and descender of the font. These two numbers are specified in the font header.

x-Height Draws the horizontal line representing the x-Height of the font. The x-Height is specified in the font header.

H-height Draws the horizontal line representing the H-height of the font. The H-height is specified in the font header.

Bezier polygon Draws the line segments connecting the control points of the Bezier curves of the character.

Figure 18-3
View Preference Dialog Box



Operator Icons

18.2.3

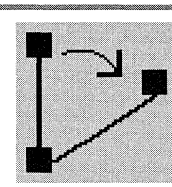
The next few sections describe the operator icons that appear to the left of the edit window. Several of these operator icons perform several actions. You can specify additional actions by holding down keys on the keyboard while clicking or releasing.

Some operations display the bounding box of the shape being manipulated and wait for you to confirm the operation. You confirm the operation by clicking on the left mouse button; you can cancel the operation by clicking on the right mouse button.

You can undo most of the operations performed by the operators by choosing Undo in the OutLine menu.

Moving Control Points

18.2.4

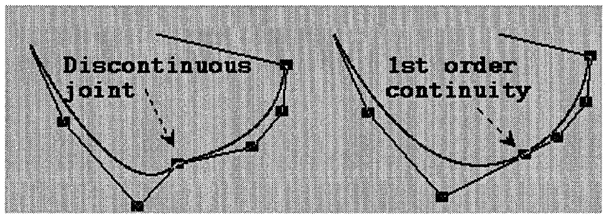


To move a control point, click on the Move Control Points operator, then point the cursor to the inside of the rectangular handle that encloses the control point. Press the left mouse button and drag the control point to where you want to put it. Where you release the button will be the new position of the control point.

When working with a low- or medium-resolution screen, it is not always possible to move a control point to an exact coordinate point, such as (100, 100). You can examine or alter the coordinate of a control point to the exact values you desire by holding down the Ctrl key and clicking on the control point handle whose coordinates you wish to change or examine. A dialog box will pop up for data entry.

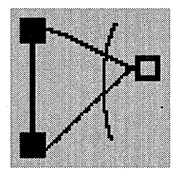
If you want a curve to flow smoothly at a segment juncture (between two Bezier curves, or between a curve and a line), hold the [Shift] key and click on the handle of the control point that joins the two segments. FEDIT will adjust the coordinates of that point to ensure a first-order continuity between the two segments. You should apply this operator only when the adjoining segments are sufficiently close to continuity. If the curve turns sharply at the control point in question, using this operator will most likely yield undesirable results.

Figure 18-4
Smoothing Lines



Deleting a Segment

18.2.5

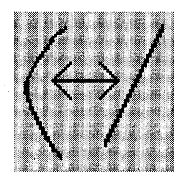


To delete a segment in a shape, click on the Delete Segment operator, then click on the control point that ends the segment you want to delete. You cannot delete a Bezier control point that is not the end point of a curve.

To merge two Bezier curves into one, hold down the [Shift] key, then click on the handle of the control point connecting the two curves. Not all pairs of curves can be merged successfully while retaining the original shapes of the two curves. If the resulting curve has a significantly different shape then the original ones, undo the change (Undo is in the OutLine menu).

Shape Mutation

18.2.6



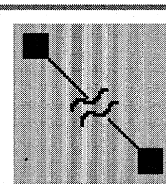
To change a line to a curve or to change a curve to a line, click on the Shape Mutation operator.

Click near the segment you wish to mutate. FEDIT shows you the bounding rectangle of that segment. Click the left button again to confirm the change or click the right button to cancel the operation.

If the segment is a line, FEDIT replaces it with a Bezier curve that has two control points along the line. If the segment is a curve, FEDIT deletes its control points; the curve's end points become the end point for the line.

Mitosis

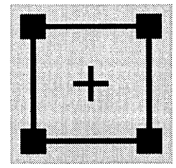
18.2.7



The mitosis operator breaks a segment into two segments of the same kind. Click near the segment you wish to duplicate, then click the left button anywhere in the segment where you want it broken, or click the right button to cancel the operation.

Adding a Rectangular Shape

18.2.8

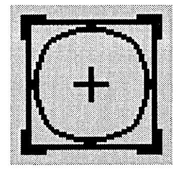


The Rectangle operator allows you to create a rectangle in the edit window. If you hold the left button while dragging, you change the size of the rectangle; if you hold down both the left and right buttons, you can move the entire rectangle around.

The rectangle consists of four line segments running in clockwise direction.

Adding an Oval Shape

18.2.9

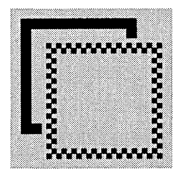


The Oval operator allows you to create an oval in the edit window. While you click and drag the mouse, FEDIT draws a rectangle. When you release, FEDIT creates an oval that consists of four Bezier curve segments running in the clockwise direction.

The oval is bounded by the rectangle you saw while dragging.

Shape Transformation

18.2.10



The Shape Transformation operator is the most complex and versatile of the group. You use this operator to duplicate, move, scale, rotate, or shear a group of shapes.

To use the Shape Transformation operator:

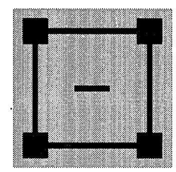
1. Drag out a rectangle that encloses all the shapes you want to alter. The bounding rectangle that encloses all the selected shape will be highlighted. As usual, you can cancel the operation at this point by clicking the right button. Note that in order for a shape to be selected, all the control points in that shape must fall into the rectangle.
2. At this point, you choose whether you want to perform operations on the selected shapes or on a duplicate of the selected shapes. If you hold down the Shift key, FEDIT makes a copy of the selected shapes before performing subsequent transformations. If you do not hold down the Shift key, all transformations will be applied to the selected shapes.
3. To move the selected shapes, click the left mouse button and move the mouse; the shapes will move with the mouse. Release the button when you have positioned the shapes where you want them. As described above, you can copy the shapes by holding the Shift key while clicking and dragging.
4. To scale the selected shapes, hold the Ctrl key down while dragging out a scaling rectangle. The selected shapes will be scaled to fit into the rectangle you dragged out.
5. To rotate the selected shapes, hold the Alt key and the left mouse button while dragging with the mouse. You move the mouse in a motion reminiscent of traversing a circle around the center of the rectangle bounding the selected shapes.

6 To shear the selected shapes in the x-direction, hold the left arrow ← or right arrow → key while you hold the left button and drag.

7 To shear the selected shapes in the y-direction, hold the up arrow ↑ or down arrow ↓ key while you hold the left button and drag.

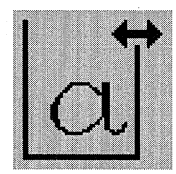
While dragging out a rectangle or controlling the angle of rotation or shear, the intermediate transformed bounding rectangle (in the case of rotation or shearing, a bounding polygon) can be moved around by holding both the left and right mouse buttons down.

Deleting a Shape 18.2.11



To delete a shape, click on the Delete operator, then drag out a rectangle that encloses the shapes you want to delete. The shapes about to be deleted will be highlighted. Click the left button to confirm deletion or the right button to cancel the operation. Again, all control points of a shape must fall into the enclosing rectangle to be selected for deletion.

Changing the Setwidth 18.2.12

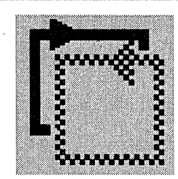


To change the setwidth for a character, click on the SetWidth operator. When you click on the SetWidth operator, FEDIT rescales the edit window so that the entire character and the setwidth line are visible.

To move the setwidth line, click inside the setwidth line's control handle (at the base line) and drag it along the x-axis.

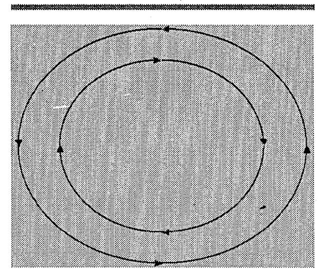
You can also change the setwidth directly by entering the number in the Info dialog in the Character menu.

Viewing and Altering the Winding Direction 18.2.13



To view or alter the winding direction, click on the Winding Direction operator.

When you select this icon, FEDIT redraws the edit window to show the winding directions of the shapes (with arrows running along the shapes instead of the usual control point handles).

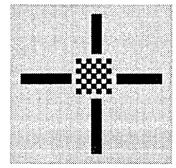


To reverse the winding direction in a shape, click near the shape. FEDIT highlights the selected shape. Click the left button again to confirm the change or the right button to cancel the selection.

In general, if one shape is totally enclosed within another, you should make sure that the outer shape winds in the counter-clockwise direction while the inner one winds in the clockwise direction, as shown in the figure on the right. The winding direction is very important in the shape merging operator (described next).

Merging Shapes

18.2.14



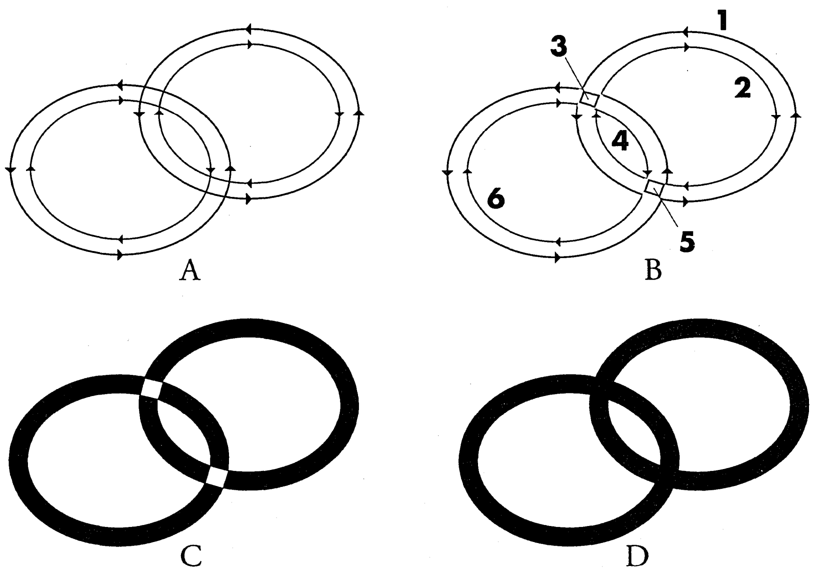
You use the Merge operator to merge two or more intersecting shapes. To merge shapes, drag out a rectangle that encloses all the shapes you want to merge. FEDIT highlights the bounding rectangle of the selected shapes. Click the left button to confirm or the right button to cancel the operation.

If you create two or more intersecting shapes, you will notice that the odd-even rule of filling causes the areas where the shapes intersect to be filled with white, rather than black. This is usually not desireable.

The Merge operator allows you to merge intersecting shapes so the resulting shapes have the same outline, but no longer intersect.

In Figure 18-4, the original shapes (A) consist of two pairs of intersecting concentric circles (four shapes). When these shapes are merged, FEDIT creates six shapes (B): the shape labeled 1 traces the arcs of the two outer-most circles; the other five shapes trace parts of the inner circles.

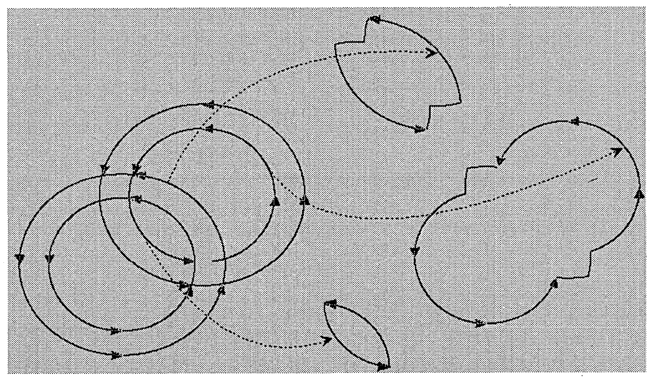
Figure 18-5
Merging Shapes



Shape merging is useful in eliminating intersecting regions where pixels are not rendered by the even-odd rule of filling. In Figure 18-5 above, the interlocked concentric circles will yield (C). By merging the shapes and then deleting the shapes labelled 3 and 5 in (B), we obtain (D).

To illustrate the importance of winding directions in shape merging, suppose we alter the winding directions of the original concentric circles such that all four circles wind in the clockwise direction, shown in Figure 18-6. When we merge these shapes, FEDIT produces only four merged shapes. The outer one is the same as that of the previous merging, but the inner ones are quite different.

Figure 18-6
Winding Direction and Merging

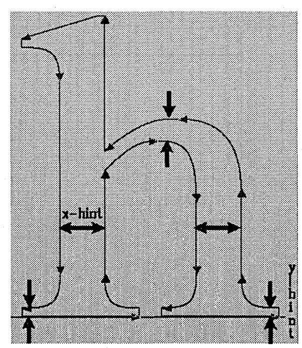


Editing Hints

18.3

When characters are rendered at low resolution, they often suffer from pixel dropout or disproportionate weights on strokes. A 10-point, 72 dpi screen font is considered a low resolution font. A **hint** is data structure that helps low resolution rendering to minimize these problems.

Each character can have numerous hints associated with it. The proprietary hinting technology used in FEDIT is licensed from Digital Typeface Corporation. While we cannot document the hinting technology in full detail, we can explain enough to allow you to use the hint editing features of FEDIT.



- A hint controls the line widths of horizontal and vertical lines, or optionally, the line widths of curves where the tangent is horizontal or vertical. An x-hint controls x-coordinates (that is, the width of vertical lines). A y-hint controls y-coordinates (horizontal line width). Thus a character can have as many hints as there are horizontal and vertical line segments pairs (remember that a rendered "line" is actually bounded by two segments).
- A hint data structure has position, range, and length components. A hint affects all shapes that fall within the band defined by its range. The length component specifies the width of the shape when the font is rendered.

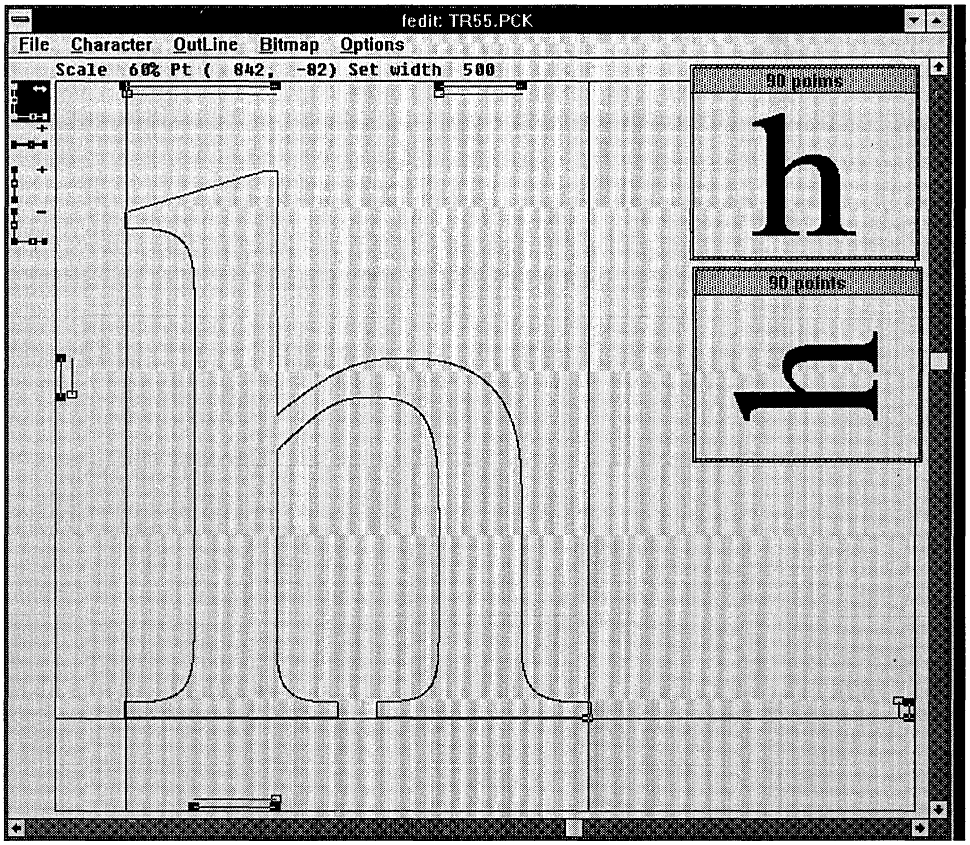
The Hint Editing Window

18.3.1

You bring up the hint editing window by choosing the Hint command under the OutLine menu. Figure 18-7 shows the hint editing window.

In the hint editing window, the shape operator icons are replaced by the hint operator icons. The control points or arrows on the character are not displayed in the hint editing window.

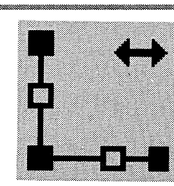
Figure 18-7
Hint Editing Window



The x-hints with decreasing coordinates appear along the top edge of the edit window; x-hints with increasing coordinates appear along the bottom edge. The y-hints with decreasing coordinates appear along the left edge; y-hints with increasing coordinates appear on the right edge. See the explanation of hint coordinates below.

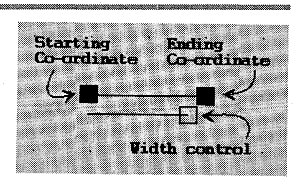
Altering a Hint

18.3.2



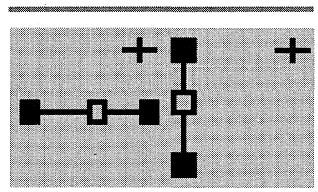
Each hint has three control handles that represent the starting and ending coordinate values of the hint (solid rectangles) and the width control (hollow rectangle).

To alter a hint, click on the Alter Hint operator, then point the cursor at one of the control handles and drag.



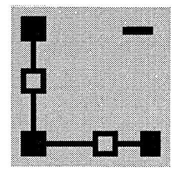
Hints are effective only when rendering at low resolution. To view the effects of the hints, you should scale the sample windows to about 10 or 12 points. In addition, turn on the AutoRedraw option under the Option menu. The sample windows will continuously redraw as you modify the hints. Turn on the discarded option only when the sample windows are small; larger-sized characters take longer to render and affect the response of the mouse in altering hints.

Creating a Hint 18.3.3



To create a hint, click on either the x- or y-hint operator. Then click in the edit window to create the hint. If you're creating an x-hint, the x-coordinate of the mouse at the time you click on the left button will be starting position of the hint. If you are creating a y-hint, the y-coordinate of the mouse will be the starting position.

Deleting a Hint 18.3.4



To delete a hint, click on the Delete hint operator, then click on the hint you want to delete. FEDIT highlights the selected hint. Click the left button again to confirm the deletion or the right button to cancel the operation.

Editing Bitmaps 18.4

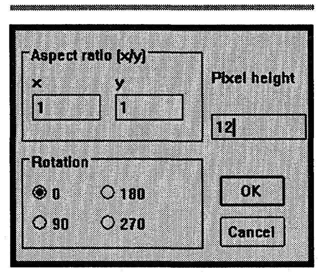
To obtain ultimate fidelity of a font at low resolution, a hand-tuned bitmap is unavoidable. In the PenPoint operating system, a bitmap (or bitmaps) is considered an integral part of a font, tied closely to the rest of the font's data structures. It is not an entity that can exist independently (that is, there is no provision for a bitmap font without outline data). Additionally, PenPoint has no mechanism by which you can attach an externally created bitmap to an outline font. You must use FEDIT to create and render a bitmap initially, then edit it to your heart's content.

Because there is a close binding between the outline metrics (in font units) and bitmap pixel metrics (in pixel units), creating and editing bitmaps should be considered finishing touches in the font creation process.

The worst thing you can do is to alter or create a character so that its vertical metrics exceed the current maximum ascender or minimum descender of a font for which you have already built bitmaps. FEDIT is currently not very flexible in regard to support this kind of activity. However, FEDIT does provide character cell alteration tools (described below) to handle this scenario. You won't have to start building the bitmaps from scratch, but it is still a tedious process which should be avoided.

Creating a Bitmap 18.4.1

To create a bitmap, choose the Create command under the Bitmap menu. A dialog box pops up for entering the specification of the bitmap:



Aspect ratio Enter the aspect ratio of the device for which the bitmap font is being created. Use small integers. For example, most nine-pin dot matrix printers have an aspect ratio of 2:1; enter x=2 and y=1.

Pixel Height Enter the number of scanlines per 1000 font units. For example, for a 12-point font on a 300 dpi printer, this number will be (12 / 72) * 300 = 50 pixels.

Rotation Specify any one of the four rotations. If you are creating bitmaps with 1:1 aspect ratio, you need only to create the bitmap with zero rotation, even if you are planning to use your font in more than one rotation. The PenPoint font subsystem will perform rotation for you automatically. Angles are measured counter-clockwise from the x-axis.

To maintain consistency between bitmaps and outline data, you should note the following:

- Altering the shapes of a character does not affect any existing bitmaps. To update the existing bitmaps to the character's new appearance, select that character in pixel editing mode (described below) and execute the Re-render Char command under the Bitmap menu. Repeat this procedure for each bitmap you have created.
- If you altered the height of a character to the extent that the character no longer fits the height of the bitmap (the height of a bitmap is the y-dimension for 0 or 180 rotation, x-dimension for 90 or 270), you would have to manually add rows (or columns) using the Edit Char command (described below).
- If you delete a character from the font, FEDIT automatically deletes the bitmap cell for that character from each existing bitmap.
- If you add a character to the font, FEDIT appends a bitmap cell for that character to the end of each bitmap. The width of each cell is computed based on the initial setwidth you assigned to the character. After you have added shapes to the new character, edit that character for each bitmap in pixel editing mode and execute the Re-render Char command. Because a new character did not have any shape at the time it was created, no pixel was drawn to the new character cells and it will be invisible. Click on the blank space just past the end of the character list to select the new character. The bitmaps will be sorted in ascending order of AFII numbers when they are saved. Thus, after you save and re-open the font file, the new character will appear in a new place.

Deleting a Bitmap 18.4.2

To delete a bitmap, choose Edit from the Bitmap menu. Select a bitmap from the list box. Click on the Delete button. Once you delete a bitmap, it cannot be undone; think twice before you do it.

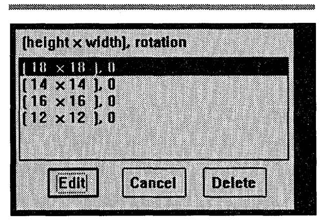
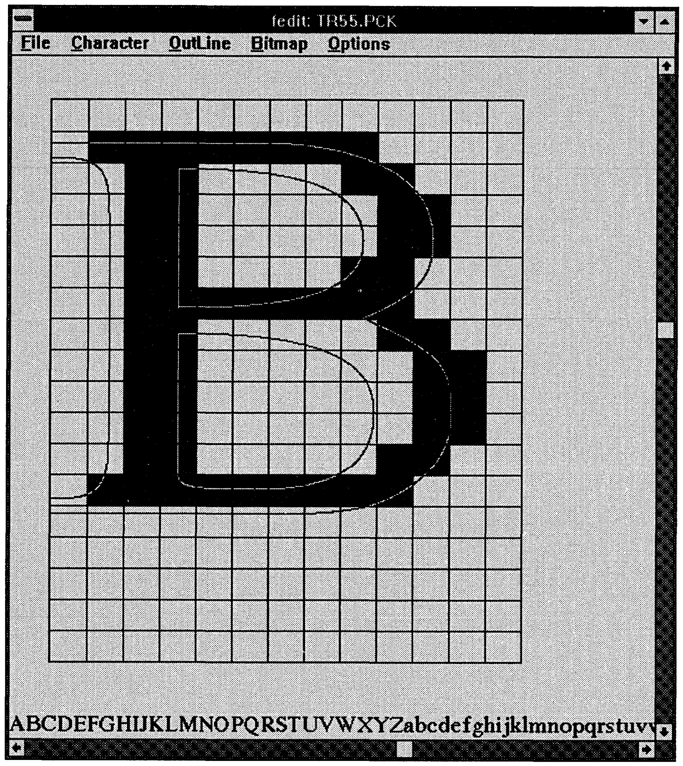


Figure 18-8
Bitmap Edit Window



Pixel Editing

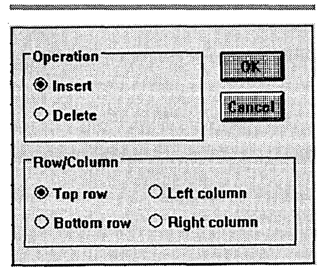
18.4.3

To edit a bitmap, choose Edit from the Bitmap menu. Select the bitmap you want to edit and click on the Edit button. The bitmap edit window will appear. The list of characters in the bitmap is shown on the bottom of the main window. You can scroll the character list with the horizontal scroll bar. If the bitmap you are editing is rotated 90 or 270, the character list appears on the right edge of the main window and you scroll it with the vertical scroll bar instead.

Select the character you want to edit by clicking on that character in the list. Click on a pixel cell in the pixel grid of the character to invert its state. FEDIT updates the character image in the character list instantly so you can monitor its appearance as you modify the pixels.

The initial state of the bitmap is obtained by rendering each character in the font from the outline data.

Altering Cell Dimensions 18.4.4



To alter the dimension of a character cell, choose Edit Cell from the Bitmap menu. You may delete or insert a border row or column.

A bitmap with a 0° or 180° rotation is said to be row major (that is, every character in the bitmap has the same number of rows). If you add or delete a row, the entire bitmap will be affected. A warning message will appear to ask for confirmation. Changing the number of columns will affect only the character you are editing.

A bitmap with a 90° or 270° rotation is said to be column major. Changing the number of columns will affect the entire bitmap while changing the number of rows will affect only the character you are editing.

In general, you should not alter the dimensions of a character by more than one pixel (if any at all). Under some circumstances, the font subsystem in PenPoint computes a character's pixel metrics by scaling its metrics in font units. Fidelity problems can occur if the scaled metrics differ too greatly from the actual metrics in the bitmap.

Miscellaneous Functions 18.5

Copying to the Clipboard 18.5.1

You can transfer shape and hint data (but not bitmap data) using the Windows clipboard. You must be in the shape editing mode to enable copying.

Choose Copy All from the OutLine menu. All the shape and hint data will be copied to the clipboard.

You may also copy a portion of the character by using one of three methods:

- Choose Selective Copy from the OutLine menu.
- Drag out a rectangle in the shape editing window. All the shapes and hint data enclosed in this region are now selected for copying. The bounding rectangle for the selected shapes and hints will be highlighted.
- Click the left button to confirm the operation; click the right button to cancel it.

Pasting from the Clipboard 18.5.2

Pasting is enabled if there are shape or hint data in the clipboard; you must be in the shape editing or hint editing mode.

Choose Paste Absolute if you want to copy the clipboard data to the current character in the same position from which the data were copied.

Choose Paste Relative if you want to copy the clipboard data to a new position in the current character. In this mode, after you have selected the operation from the menu, press the left mouse button in the editing window. The bounding rectangle for the clipboard data will appear. Move this rectangle to the location where you want to place the shapes or hints. Release the button.

You can undo a paste operation.

Subset Saving 18.5.3

You can save a subset of the characters and bitmaps in a font to a different file. Choose Save Subset from the File menu. FEDIT prompts for the output file name as well as a file containing the list of characters and bitmaps you wish to save. The format of this file is a list of AFII numbers and bitmap specifications. AFII numbers are simply signed integers. Bitmap specification starts with a colon (:) followed by the width, height, and rotation of the bitmap. For example, the following list saves the characters A to E and the bitmap with 12 x 12, 0 rotation:

    501 502 503 504 505 :12 12 0

The order of the entries in the list is unimportant and new lines are permitted in the file. The saved bitmaps will contain only characters you have chosen in the list.

Examining and Editing the Font Header 18.5.4

To examine and edit the global information about the font, choose the Font Header or the Font Attribute command under the File menu. You can modify any fields that have a border; fields without a border are maintained by FEDIT.

Table 18-1 describes the fields in the Font Header window; Table 18-2 describes the fields in the Font Attribute window. All metrics are in font units.

Figure 18-9
Font Header Window

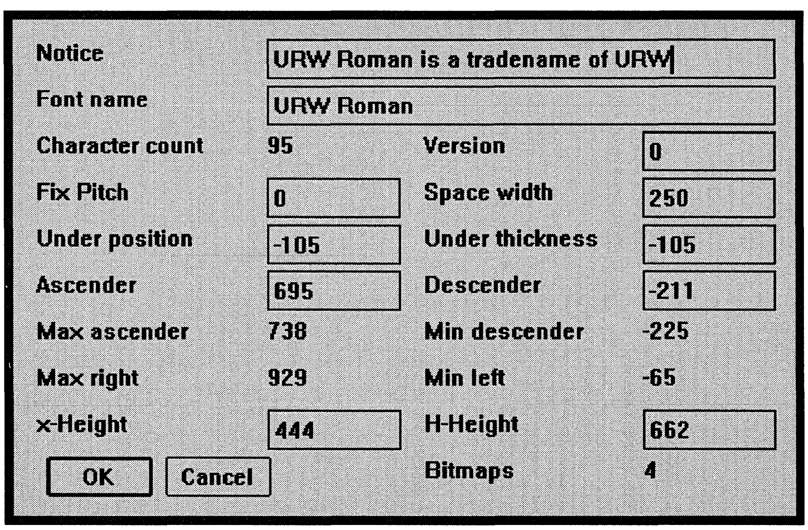


Table 18-1
Font Header Window Fields

Field	Description
Notice	Copyright notice.
Font name	Name of the font.
Character count	Number of characters in this font.
Version	Version number of the font.
Fixed pitch	0 if the font is proportionally spaced; character width if the font is monospaced.
Space width	Width of the space character.
Under position	Offset from the base line where an underline will be placed. This is a negative number.
Under thickness	Width of an underline.
Ascender	A positive number giving the y-offset of the ascender line. Typically, the top of a capital letter. This value is the opinion of the font designer, and does not necessarily correspond to the tallest character in the font.
Descender	A negative number giving the y-offset of the descender line. Typically, the bottom of a j. This value is the opinion of the font designer, and does not necessarily correspond to the deepest descender in the font.
x-height	A positive number giving the y-offset of the x-height line. Typically, the top of a x. This value is the opinion of the font designer, and does not necessarily correspond to the actual x.
h-height	A positive number giving the y-offset of the h-height line. Typically, the top of an h. This value is the opinion of the font designer, and does not necessarily correspond to the actual h.
Max ascender, Min descender, Max right, Min left	The smallest and largest x and y values found in the font. These can be used to construct a rectangle that would enclose any character from the font. These values are not the opinion of the font designer; they are computed automatically from the character definitions and cannot otherwise be edited.
Bitmaps	The number of bitmaps in the font.

Figure 18-10
Font Attribute Window

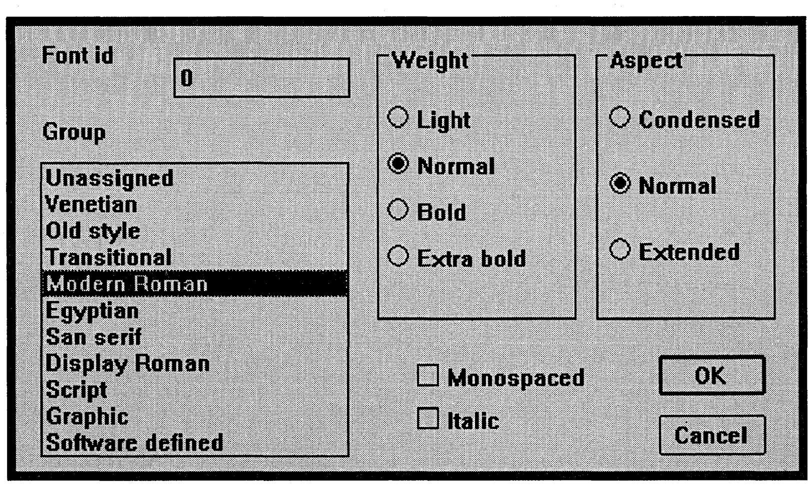


Table 18-2
Font Attribute Window Fields

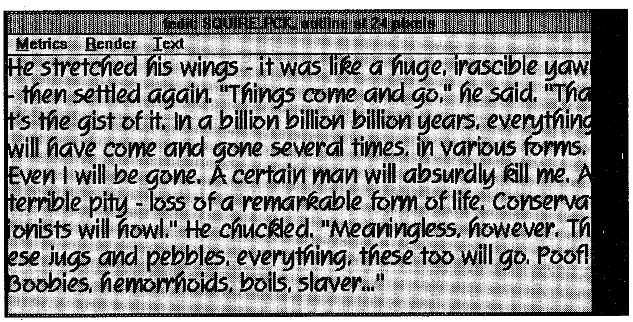
Field	Description
Font ID	A 118-bit number assigned by GO.
Group	The group characterizes the font design; this is used by software to find a "best match" when requested font is unavailable.
Weight	Specifies the stroke weight of the font.
Aspect	Specifies the designed aspect ratio (not the device aspect ratio) of the font.
Monospaced	Check this box if the font is designed to be a monospaced font.
Italic	Check this box if the font is an italic font.

Examining Text Samples

18.5.5

To view a sample of ASCII text rendered from the font, choose Show Sample from the Character menu. FEDIT displays a text sample window. Choose the same command again to take down the window.

Figure 18-11
Test Sample Dialog Box



To render a sample, choose the Metrics menu; a dialog similar to the bitmap creation dialog pops up. Enter the text size you want to view. The title bar of the sample window shows whether the text sample came from a bitmap or was rendered from outline.

If you subsequently alter the source (bitmap or outline) of the sample, choose Render to update the sample.

You can specify the text sample by selecting the File command from the Text menu and entering a file name containing the text you want to view. The Default command shows the text sample illustrated above.

Using Fonts in Documentation

18.5.6

If you need use your FEDIT font in printed documentation, you should use FEDIT to save your font as an Adobe Type I font (described below).

If you are printing to a PostScript printer, you can download the font to the printer or, depending on your word processor, you can configure the font in the word processor and enable it to download the font.

If you are printing to a PCL printer, you need to use a word processor or page-layout program that uses Adobe Type Manager under either Microsoft Windows or Macintosh. When you use Adobe Type Manager, your font will be rendered on screen. When you print to a PCL printer, Adobe Type Manager creates PCL bitmaps for your font and downloads them to the printer.

Adobe Type I Fonts 18.6

An Adobe Type I font is a Postscript program; as such, a robust reading of such a font requires an implementation of a Postscript interpreter. FEDIT does not implement such an interpreter. Rather, it employs heuristics based on common conventions described in the book *Adobe Type 1 Font Format* published by Adobe Systems, Inc. One can construct a perfectly legitimate Adobe Type I font file without regard to these conventions. GO does not guarantee that FEDIT will load all Adobe Type I font files successfully. In addition, FEDIT currently imposes a size limit of 64K on such a font file.

Certain incompatibilities exist between the hinting technologies of Adobe Systems Inc. and Digital Typeface Corporation. Where an Adobe hint feature does not apply to PenPoint, it is discarded. Thus, there may a loss of information in loading an Adobe Type I font.

Additionally, each character in an Adobe font has a name either defined implicitly by standard encoding, or explicitly by a custom encoding. FEDIT, on the other hand, identifies a character by its AFII number. When FEDIT loads an Adobe font, it maps an Adobe character by its character code (0-255) to an AFII number according to Table 18-3, below. The character names are discarded.

Saving an Adobe Font 18.6.1

Additional information is necessary to create an Adobe Type I font. The dialog box show here pops up when you attempt to save the currently opened font as an Adobe font. Table 18-3 describes the fields in the dialog box. The description assumes a certain familiarity with the PostScript language.

Figure 18-12
Save Font Dialog Box

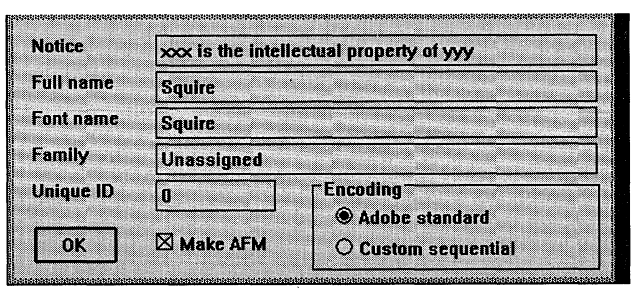


Table 18-3
Adobe Font Fields

Field	Description
Notice	Copyright notice of the font. The /Notice entry in the created /FontInfo dictionary is set to this string.
Full name	Full name of the font. The /FullName entry in the created /FontInfo dictionary is set to this string.
Font name	Name of the font. The /FontName entry in the created font dictionary is set to this string. This is the name you would use with the findfont operator in your PostScript program.
Family	The /FamilyName entry in the created /FontInfo dictionary is set to this string.
Unique ID	The /UniqueID entry in the created font dictionary is set to this string.
Make AFM	Instructs FEDIT to output an Adobe font metrics file also. The extension of the metric file is always set to .AFM and .PFB for the font body.
Encoding	Choose the encoding of the font. If you choose the Adobe standard encoding, FEDIT checks that the AFII number of each character in the font has a mapping in the Adobe encoding. If any character fails the test, an error message will appear and the file will not be saved. In addition, the Adobe standard encoding can hav a maximum of 256 characters only. The custom sequential encoding is primarily for a font with graphics symbols that don't have standard AFII to Adobe mappings. In this mode, each character in the font, in ascending AFII numbers, is assigned to the standard 96-character ASCII encoding, starting with the space character. Thus, you cannot encode a font with more than 96 characters in this mode.

Adobe Standard Encoding to AFII Mappings 18.6.2

Table 18-4 shows the mapping between an Adobe character and its assumed AFII code in the Adobe standard encoding mode. The column labeled "encoded?" simply indicates whether a character is designated to be encoded or not according to the Postscript specification. FEDIT does not pay attention to this property.

Table 18-4
Mapping of Adobe Standard Encoding to AFII Codes

Adobe Character Code	Adobe Character Name	AFII Code
32	space	32
33	exclam	33
34	quotedbl	34
35	numbersign	35
36	dollar	164
37	percent	37
38	ampersand	38
39	quoteright	39
40	parenleft	40
41	parenright	41
42	asterisk	42
43	plus	43

continued

Table 18-4 (continued)

Adobe Character Code	Adobe Character Name	AFII Code
44	comma	44
45	hyphen	45
46	period	46
47	slash	47
48	zero	48
49	one	49
50	two	50
51	three	51
52	four	52
53	five	53
54	six	54
55	seven	55
56	eight	56
57	nine	57
58	colon	58
59	semicolon	59
60	less	60
61	equal	61
62	greater	62
63	question	63
64	at	64
65	A	65
66	B	66
67	C	67
68	D	68
69	E	69
70	F	70
71	G	71
72	H	72
73	I	73
74	J	74
75	K	75
76	L	76
77	M	77
78	N	78
79	O	79
80	P	80
81	Q	81

continued

Table 18-4 (continued)

Adobe Character Code	Adobe Character Name	AFII Code
82	R	82
83	S	83
84	T	84
85	U	85
86	V	86
87	W	87
88	X	88
89	Y	89
90	Z	90
91	bracketleft	91
92	backslash	92
93	bracketright	93
94	asciicircum	94
95	underscore	95
96	quoteleft	96
97	a	97
98	b	98
99	c	99
100	d	100
101	e	101
102	f	102
103	g	103
104	h	104
105	i	105
106	j	106
107	k	107
108	l	108
109	m	109
110	n	110
111	o	111
112	p	112
113	q	113
114	r	114
115	s	115
116	t	116
117	u	117
118	v	118
119	w	119

continued

Table 18-4 (continued)

Adobe Character Code	Adobe Character Name	AFII Code
120	x	120
121	y	121
122	z	122
123	braceleft	123
124	bar	124
125	braceright	125
126	asciitilde	126
161	exclamdown	161
162	cent	162
163	sterling	163
164	fraction	164
165	yen	165
166	florin	166
167	section	167
168	currency	168
169	quotesingle	169
170	quotedblleft	170
171	guillemotleft	171
172	guilsinglleft	172
173	guilsinglright	173
174	fi	174
175	fl	175
177	endash	177
178	dagger	178
179	daggerdbl	179
180	periodcentered	180
182	paragraph	182
183	bullet	183
184	quotesinglbase	184
185	quotedblbase	185
186	quotedblright	186
187	guillemotright	187
188	ellipsis	188
189	perthousand	189
191	questiondown	191
193	grave	193
194	acute	194
195	circumflex	195

continued

Table 18-4 (continued)

Adobe Character Code	Adobe Character Name	AFII Code
196	tilde	196
197	macron	197
198	breve	198
199	dotaccent	199
200	dieresis	200
202	ring	202
203	cedilla	203
205	hungarumlaut	205
206	ogonek	206
207	caron	207
208	emdash	208
225	AE	225
227	ordfeminine	227
232	Lslash	232
233	Oslash	233
234	OE	234
235	ordmasculine	235
241	ae	241
245	dotlessi	245
248	lslash	248
249	oslash	249
250	oe	250
251	germandbls	251

Font File Formats 18.7

This section describes the two of the three font file formats supported by FEDIT. For information on the Adobe Type I format, see *Adobe Type I Font Format* published in 1990 by Adobe Systems, Inc.

The font file formats described here are:

- Nimbus-Q format
- PenPoint Packed Format

The Nimbus-Q Format 18.7.1

The Nimbus-Q font file has four main sections illustrated below:

Font Header 18.7.1.1

The Nimbus-Q font header has the following structure:

```
typedef struct NIMBUSQ_HDR {
    long        signature;
    char        notice[80];
    char        fullName[80];
    char        fontName[80];
    char        family[80];
    char        weight[80];
    char        version[80];
    char        charSet[80];
    double      italicAngle;
    short       fixPitch;
    short       spaceWidth;
    short       underPosition;
    short       underThickness;
    short       hHeight;
    short       xHeight;
    short       ascender;
    short       descender;
    short       nChars;
} NIMBUSQ_HDR;
```

The font file signature is not under the control of GO. The currently known value is 1314310_{10}.

AFII Number Array 18.7.1.2

The AFII number array holds the character IDs in the font. Each number is a 16-bit integer. The size of the table is specified by the **nChars** field in the font header.

Character Data File Positions 18.7.1.3

The character data file positions area is table of 32-bit file positions pointing to the character data. There are as many entries in the table as there are characters in the font.

Character Data 18.7.1.4

The start of each character data block is located by the file position table above. Each character data block consists of the following fields:

- Setwidth of the character (16-bit word).
- The x- and y-coordinates of the lower-left-hand corner of the character's bounding rectangle (two 16-bit words).
- The x- and y-coordinates of the upper-right-hand corner of the character's bounding rectangle (two 16-bit words).
- Number of x-hints in the character (16-bit word).
- As many x-hint structures as the number in the preceding field. Each hint contains a starting coordinate, an ending coordinate, and a length (3 16-bit words).

- Number of y-hints in the character (16-bit word).
- As many y-hint structures as the number in the preceding field.
- Next comes the segment data. Each segment starts with a 16-bit code, followed by 0, 1, or 3 pairs of coordinate values, depending on the shape code:
    - **0** Move to the coordinate point that follows.
    - **1** Draw a line to the coordinate point that follows.
    - **2** Draw a curve to the 3 coordinate points that follow.
    - **3** End of shape—no data follows this code.

The PenPoint Packed Format 18.7.2

The GO packed format is used by the PenPoint font subsystem. It is a highly compressed format suitable for the PenPoint environment but not for programmers' mental health. The following list shows the file's main layout.

Note This format will change in future releases of PenPoint.

Font Header
URW Number Array
Bitmap Directory
Character Directory
Short Bezier Dictionary Pointer
Long Bezier Dictionary Pointer
Shape Data Pointer
Hint Datra
Short Bezier Dictionary
Long Bezier Dictionary
Shape Data
Bitmap Data

Font Header 18.7.2.1

The GO font header has the following structure.

```
typedef struct FONT_HDR {
    long        signature;
    char        notice[80];
    char        fullName[80];
    short       fontId;
    short       attr;
    short       version;
    short       fixPitch;
    short       spaceWidth;
    short       underPosition;
    short       underThickness;
    short       xHeight;
    short       hHeight;
    short       ascender;
    short       descender;
    short       nChars;
    short       maxTriples;
```

```
        short       nShapeRuns;
        short       bigModel;
        short       maxAscender;
        short       minDescender;
        short       maxRight;
        short       minLeft;
        short       reserved[14];
        short       nBitmaps;
    } FONT_HDR;
```


Most of the fields in the font header have been described in a previous section. Here are descriptions of fields not documented before:

- The GO font file signature is 131431_{10}.
- The font attribute word attrib has the following structure:

group (4 bits)		**weight** (2 bits)	
Unassigned	0	Light	0
Venetian	1	Normal	1
Old style	2	Bold	2
Transitional	3	Extra bold	3
Modern roman	4	**italic (1 bit)**	
Egyptian	5	Normal	0
San serif	6	Italic	1
Display roman	7	**aspect** (2 bits)	
Script	8	Condensed	0
Graphic	9	Normal	1
Software defined	10	Extended	2
monospaced (1 bit)		**reserved** (6 bits)	
Proportional	0		
Mono	1		

maxTriples contains the number of hints (x and y) of the character which has the maximum number of hints in the font.

- The fields **bigModel** and **nShapeRuns** are explained in the shape data section.

AFII Number Array 18.7.2.2

The AFII number array holds the character IDs in the font. Each number is a signed 16-bit integer. The size of the table is specified by the **nChars** field in the font header. The array is sorted in ascending order of the AFII numbers.

Bitmap Directory 18.7.2.3

The bitmap directory contains as many entries as there are bitmaps in the font. This section can be absent if there are no bitmaps in the font. Each entry describes a bitmap and contains the following fields:

Canonical width and height The pair of values obtained by transforming (1000, 1000) by the matrix used for scaling the font when the bitmap was created (two 16-bit words).

Rotation The angle of rotation of the bitmap (16-bit word). It can be any one of the values 0, 90, 180, or 360.

SizeX, SizeY The actual width and height of the bitmap in pixel units (two 16-bit words).

Reserved Reserved by GO for future use (16-bit word).

Ascender The ascender of the bitmap in pixel units (16-bit word).

Row byte size The number of bytes in a row of the bitmap, must be an even number (16-bit word).

Bitmap file position The location in the file where the pixel map is stored (32-bit word).

Character Directory 18.7.2.4

The character directory contains as many character definition entry as there are characters in the file, plus one. The extra entry serves as a sentinel for the hint and shape indices calculations. Each entry has the following structure:

Setwidth In font units (16-bit word).

Bounding rectangle Two 24-bit packed number pairs for the lower-left and upper-right corners of the character's bounding rectangle (6 bytes).

X-hint index The index number of the starting x-hint data in the hint data section of the file (16-bit word). The number of x-hints for this character is obtained by subtracting this number from the next field (which can be 0).

Y-hint index The index number of the starting y-hint data in the hint data section of the file (16-bit word). The number of y-hints for this character is obtained by subtracting this number from the x-hint index of the next character (which can be 0).

Shape index The index number of the starting shape data in the shape data section (16-bit word). The number of shapes for this character is obtained by subtracting this number from the shape index of the next character.

There will be many occurrences of a 24-bit packed number pair in a font file. It consists of two signed 12-bit numbers, packed into 3 bytes and has the following structures:

byte 1 Low-order byte of n1.

byte 2 Low-order of n2.

byte 3 Hi-order nibble of n1, hi-order nibble of n2.

File Pointers 18.7.2.5

The next 3 fields are 32-bit file positions pointing to the short Bezier dictionary, the long Bezier dictionary, and the shape data section.

Hint Data 18.7.2.6

The hint data section contains all the hint data of the font. The hints are accessed by the hint data indices of the character definition. For example, if a character's starting index for x-hints is n and it has 3 x-hints, then hint[n], hint[n+1], and hint[n+2] are the x-hints of the character.

Each hint data structure consists of a 24-bit packed number pair, encoding the starting and ending coordinates of the hint, and a 16-bit word for the length of the hint.

Shape Data 18.7.2.7

We will skip forward to describe the shape data section before the Bezier dictionaries. The shape data section contains two main subsections:

- The offset table.
- A series of variable-length shape blocks.

The offset table contains byte offsets to shape blocks. The offset is relative to the first byte past the end of the offset table. There are **nShapeRuns** (see font header) entries in the table. Each entry is a 16-bit word if the **bigModel** flag (see font header) is **false**, 32-bit word otherwise.

Each shape block contains:

- The starting point of a shape in 24-bit packed coordinate format.
- The shape data. All shape data are relative coordinate values, that is, begin with the starting point of the shape, the next point is computed by adding the values from the shape data to the current point as you decode the shape data. All shape data are signed values regardless of their bit length.
- An array of shape codes, each 4-bit long. The shape code dictates how the shape data are accessed and the array is arranged in a reverse order, that is, the first shape code is the last entry in the array. The shape codes are:

  0 Draw relative line with the 24-bit packed coordinate pair.

  1 Draw relative line with the 8 + 8 bit coordinate pair.

  2 Draw relative horizontal line with the 16-bit coordinate.

  3 Draw relative horizontal line with the 8-bit coordinate.

  4 Draw relative vertical line with the 16-bit coordinate.

  5 Draw relative vertical line with the 8-bit coordinate.

  6 Draw relative curve with the 3 24-bit packed coordinate pairs.

  7 Draw relative curve with the 3 8 + 8 bit coordinate pairs.

  8 Draw relative curve with the 16-bit index into the long Bezier dictionary.

  9 Draw relative curve with the 16-bit index into the short Bezier dictionary.

  10 Branch off to a shape block with the 16-bit index and draw **n** segments using data from that shape block. **n** is a 16-bit value that follows the shape index. Maintain the current point when branching off, do not switch to the starting point of the referenced shape block.

  11 End of shape block. No data is associated with this code.

To decode a shape data block, use this algorithm:

- Keep two pointers p0 and p1. p0 points to the beginning of the shape block data (the byte past the starting point). p1 points to the end of the shape code array. Compute p1 by adding the difference of the byte offset of the next shape block and that of the current shape block, minus 1, to p0.
- Fetch a shape code with p1, and fetch the shape data with p0. The size of the shape data to fetch depends on the shape code, as listed in the table above. Recall a shape code is a 4-bit quantity. So the first shape code is the low-order 4 bits of the byte pointed to by p1.
- Process the data. Advance p0 with the size of the data just read. Decrement p1 by 4 bits. Repeat the process until the end-of-shape code is read.

A character can contain many shapes. You need to decode as many consecutive shape blocks as needed by the character, starting with the shape index in the character definition entry.

Short Bezier Dictionary 18.7.2.8

The short Bezier dictionary is an array of Bezier control points. Each is a triple of 8 + 8 bit coordinates. Each point in the triple is relative to the previous one and the first point of the triple is relative to the current point which you maintain as you decode the shapes.

An entry in this dictionary is accessed through an index referenced by the short Bezier dictionary shape code.

Long Bezier Dictionary 18.7.2.9

The long Bezier dictionary is an array of Bezier control points. Each is a triple of 24-bit packed coordinates. Each point in the triple is relative to the previous one and the first point of the triple is relative to the current point which you maintain as you decode the shapes.

An entry in this dictionary is accessed through an index referenced by the long Bezier dictionary shape code.

Bitmap Data 18.7.2.10

The last section of the font file contains bitmap data referenced in the bitmap directory, if there is any. The first **nChars** bytes make up the width table of the bitmap, then followed by the pixel map.

This master index indexes all five volumes in the PenPoint Software Development Kit documentation. Each page number in the master index contains a code that indicates the volume in which the page is found. The codes are:

AWG *PenPoint Application Writing Guide*

AR1 *PenPoint Architectural Reference Volume I*

AR2 *PenPoint Architectural Reference Volume 2*

UI *PenPoint User Interface Design Reference*

PDT *PenPoint Development Tools*

READER'S COMMENTS

Your comments on our software documentation are important to us. Is this manual useful to you? Does it meet your needs? If not, how can we make it better? Is there something we're doing right and you want to see more of?

Make a copy of this form and let us know how you feel. You can also send us marked up pages. Along with your comments, please specify the name of the book and the page numbers of any specific comments.

Please indicate your previous programming experience:

☐ MS-DOS ☐ Mainframe ☐ Minicomputer

☐ Macintosh ☐ None ☐ Other ______________________

Please rate your answers to the following questions on a scale of 1 to 5:

	1 Poor	2	3 Average	4	5 Excellent
How useful was this book?	☐	☐	☐	☐	☐
Was information easy to find?	☐	☐	☐	☐	☐
Was the organization clear?	☐	☐	☐	☐	☐
Was the book technically accurate?	☐	☐	☐	☐	☐
Were topics covered in enough detail?	☐	☐	☐	☐	☐

Additional comments:

____

____

____

____

____

____

____

____

____

Your name and address:

Name ____

Company ____

Address ____

City ________________________ State ______________ Zip ______________

Mail this form to:

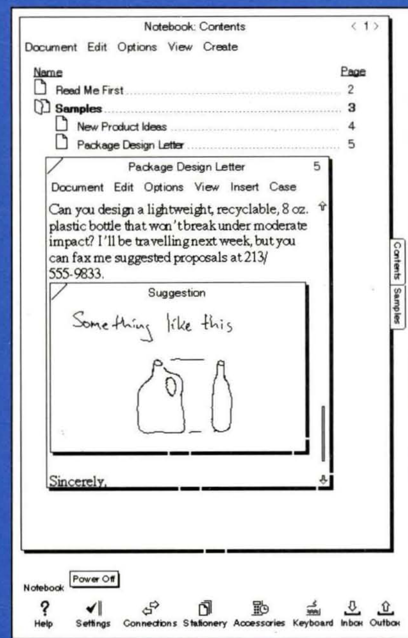
Team Manager, Developer Documentation
GO Corporation
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404–2128

Or fax it to: (415) 345-9833

Operating Systems

> $24.95 USA
> $31.95 CANADA

PenPoint™ Development Tools



PenPoint Development Tools describes the PenPoint development environment and the tools that you use to develop, test, and debug PenPoint applications. The first part introduces you to PenPoint application development, describes the organization of the PenPoint SDK documentation set, and instructs you how to run PenPoint on a 80386/486 PC. The second part describes how to debug PenPoint programs; this includes a tutorial on the PenPoint source-level debugger. The third part covers the tools provided with the PenPoint SDK, including file system tools, a bitmap editor, and a font editor.

The PenPoint operating system is a compact, 32-bit, fully object-oriented multitasking operating system designed expressly for mobile, pen computers. GO Corporation designed the PenPoint operating system as a productivity tool for people who may never have used computers before, including salespeople, service technicians, managers and executives, field engineers, insurance agents and adjustors, and government inspectors.

Other volumes in the GO Technical Library are:

PenPoint Application Writing Guide provides a tutorial on writing PenPoint applications, including many coding samples.
PenPoint User Interface Design Reference describes the elements of the PenPoint Notebook User Interface, sets standards for using those elements, and describes how PenPoint uses the elements.
PenPoint Architectural Reference, Volume I presents the concepts of the fundamental PenPoint classes.
PenPoint Architectural Reference, Volume II presents the concepts of the supplemental PenPoint classes.
PenPoint API Reference, Volume I provides a complete reference to the fundamental PenPoint classes, messages, and data structures.
PenPoint API Reference, Volume II provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

GO Corporation was founded in September 1987 and is a leader in pen computing technology for mobile professionals. The company's mission is to expand the accessibility and utility of computers by establishing its pen-based operating system as a standard.

GO Corporation

919 East Hillsdale Blvd.
Suite 400
Foster City, CA 94404

52495>
9 780201 608618
ISBN 0-201-60861-8
60861

Addison-Wesley Publishing Company

045-00210