PALM COMPUTING® PLATFORM

Palm OS® Programming Development Tools Guide

Document Number 3011-002
Print Date 2/00

CONTRIBUTORS

Written by Gary Hillerson
Production by <dot>**PS** document production services
Engineering contributions by Keith Rollin, Derek Johnson, Ken Krugler, and Jesse Donaldson.

Palm OS Programming Development Tools Guide
Document Number 3011-002
February 7, 2000

Palm Computing, Inc.
5400 Bayfront Plaza
Santa Clara, CA 95052
USA
www.palm.com/devzone

Table of Contents

About This Document

Palm OS® Programming Development Tools Guide describes the various tools you can use to help in the development of software for Palm Computing® handhelds.

Palm OS® SDK Documentation

In addition to this book, the following documents are part of the SDK:

Document	Description
Palm OS® SDK Reference	An API reference document that contains descriptions of all Palm OS® function calls and important data structures.
Palm OS® Programmer’s Companion	A guide to application programming for the Palm OS. This volume contains conceptual and “how-to” information that complements the Reference.
Palm OS® 3.0 Tutorial	A number of phases step developers through using the different parts of the system. Example applications for each phase are part of the SDK.

What This Volume Contains

This volume is designed for random access. That is, you can read any chapter in any order. You don’t necessarily have to read some before others, though the first few chapters are designed for programmers who are new to the Palm OS. The first four chapters help you learn necessary tasks and possible features for your application.

Here is an overview of this volume:

- Chapter 1, "Using the Palm OS® Emulator." Describes Palm OS Emulator, the emulator program that you can use to test and debug your Palm OS® programs.
- Chapter 2, "Using Palm Debugger." Provides an introduction to the Palm Debugger, which is an assembly language and limited source code level debugger for Palm OS programs. This chapter describes how to use Palm Debugger, including a description of its expression language and a variety of debugging strategies and tips.
- Chapter 3, "Palm Debugger Command Reference." Provides a complete reference description for each command available in Palm Debugger.
- Chapter 5, "Using the Palm Simulator." Describes the Palm Simulator program, which you can use to simulate program execution on Macintosh computers.
- Appendix A, "Debugger Protocol Reference." Describes the API for sending commands and responses between a debugging host, such as Palm Debugger, and a debugging target, which can be a Palm Computing handheld ROM or an emulator program such as the Palm OS Emulator.
- Appendix B, "Host Control API." Describes the host control API, which provides functions that an emulated application can use to call into Palm OS Emulator for certain services.
- Appendix C, "Simple Data Types." Describes the simple data type name changes made in recent versions of the Palm OS software.

Conventions Used in This Guide

This guide uses the following typographical conventions:

This style...	Is used for...
`fixed width font`	Code elements such as function, structure, field, bitfield.
fixed width underline	Emphasis (for code elements).
bold	Emphasis (for other elements).
blue and underlined	Hot links.

1

Using the Palm OS® Emulator

This chapter describes how to use the Palm OS® Emulator program, a hardware emulator for the Palm Computing® platform. You can use the Palm OS Emulator to test and debug programs that you have developed for this platform.

NOTE: The Palm OS Emulator has previously been referred to as POSE or Poser. The name Palm OS Emulator is used throughout this book and in new versions of other Palm documentation. In this chapter, Emulator is sometimes used as an abbreviated form of Palm OS Emulator.

About the Palm OS Emulator

The Palm OS Emulator is a hardware emulator program for the Palm Computing platform, which means that it emulates the Palm hardware in software, providing you with the ability to test and debug Palm OS software on a Macintosh, Unix, or Windows-based desktop computer.

When you run a Palm OS application with the Palm OS Emulator on your desktop computer, the Palm OS Emulator fetches instructions, updates the handheld screen display, works with special registers, and handles interrupts in exactly the same manner as does the processor inside of Palm Computing platform handhelds. The difference is that the Palm OS Emulator executes these instructions in software on your desktop computer.

The Palm OS Emulator displays an on-screen image that looks exactly like a Palm™ connected organizer, as shown in Figure 1.1. You can select which type of Palm handheld device you want to emulate, and you can also specify that you want the Palm OS Emulator to display the screen in double size, which continues to provide an accurate representation and makes the Palm screen easier to view.

Figure 1.1 The Palm OS Emulator display

You can use the mouse on your desktop computer just as you use the stylus on a Palm connected organizer. You can even use the Graffiti® power writing software with Palm OS Emulator and your mouse. And Palm OS Emulator includes additional keyboard shortcuts that you can use on your desktop computer. These shortcuts are described in Using the Hardware Buttons.

You can use the Palm OS Emulator to perform some debugging of your applications, and you can use the Emulator with Palm Debugger to perform extensive debugging of your applications. When you connect the Emulator with Palm Debugger, you can debug in exactly the same manner as debugging with your application running on an actual hardware handheld device.

Standard Device Features

Palm OS Emulator accurately emulates Palm Computing platform hardware devices, and includes the following features:

- an exact replica of the Palm device display, including the silkscreen and Graffiti areas
- emulation of the Palm stylus with the desktop computer pointing device
- emulation of the Palm device hardware buttons, including:
  - power on / off button
  - application buttons
  - up and down buttons
  - reset button
  - HotSync® button
- ability to zoom the display for enhanced readability and presentation
- screen backlighting
- communications port emulation for modem communications and synchronizing

Extended Emulation Features

Palm OS Emulator also provides the following capabilities on your desktop computer that extend the standard Palm device interface.

- ability to enter text with the desktop computer
- configurable memory card size, up to 8MB

Debugging Features

Palm OS Emulator provides a large number of debugging features that help you to detect coding problems and unsafe application operations. Palm OS Emulator includes the following debugging features and capabilities:

- ability to use an automated test facility called Gremlins, which repeatedly generates random events

- support for external debuggers, including Palm Debugger, the Metrowerks CodeWarrior debugger, and gdb.
- monitoring of application actions, including various memory access and memory block activities
- logging of application activities, including events handled, functions called, and CPU opcodes executed by the application
- profiling of application performance

For more information about testing and debugging applications with Palm OS Emulator, see Testing and Debugging With Palm OS Emulator.

Using ROM Images

To run Palm OS Emulator, you need to transfer a ROM image to it. The ROM image contains all of the code used for a specific version of the Palm OS. You can obtain ROM images for different Palm OS versions from the Palm developer zone web site, or you can tell Palm OS Emulator to download the ROM from a handheld that has been placed in the device cradle and connected to the desktop computer. For more information about transferring a ROM image to Palm OS Emulator, see Loading ROM Images.

When you download a ROM image from the Palm web site, you can obtain a debug ROM image, which contains information that Palm OS Emulator uses to help you debug Palm OS applications. For more information about the debugging capabilities in Palm OS Emulator, see Testing and Debugging With Palm OS Emulator.

Downloading and Running Palm OS Emulator

You run Palm OS Emulator just like you would any other program. When Palm OS Emulator starts up, it displays an image of a handheld device, as shown in Figure 1.1.

NOTE: The first time that you start the Emulator, it does not display an image of a handheld device; instead, it asks you to create a new session. After you have defined a session configuration, the Emulator creates a new session based on those settings when it launches.

You can then use the keyboard and mouse to interact with the emulated device, as described in The Palm OS Emulator User Interface, and use the menus to interact with Palm OS Emulator.

Palm OS Emulator Runtime Requirements

The Palm OS Emulator requires one of the following runtime environments:

- Windows 98
- Windows 95
- Windows NT
- MacOS 7.5 or later
- Unix: some versions, including Linux

The Emulator is a multi-threaded 32-bit program. It does not run on Windows 3.1, even with Win32s installed.

Downloading Palm OS Emulator

The most recent released version of Palm OS Emulator for both the Macintosh and Windows is always posted on the Internet in the Palm developer zone:

http://www.palm.com/devzone

Follow the links from the developer zone main page to the emulator page to retrieve the released version of the Emulator. If you want to test-drive the version of Palm OS Emulator that is currently under development, you can follow links from the developer zone page to the Emulator seed page.

The Palm OS Emulator package that you download includes the files shown in Table 1.1.

Table 1.1 Files included in the Palm OS Emulator package

File name	Description
`Binder.exe` (Windows NT)	A program that binds the Palm OS Emulator with a ROM image for kiosk and demonstration purposes.
`Emulator.exe` (Windows) `Palm OS Emulator` (Macintosh) `pose` (Unix)	Main Palm OS Emulator executable
`Emulator_Profile.exe` (Windows) `Palm OS Emulator - Profile` (Macintosh)	Palm OS Emulator with added profiling facilities
`Docs` (directory)	Palm OS Emulator documents, including: • `_ReadMe.txt`, which describes the files in the `Docs` directory • `_News.txt`, which describes changes in the most recent version • `_OldNews.txt`, which describes previous version changes
`ROM Transfer.prc`	Palm OS program to send the Palm.ROM file to your desktop.
`HostControl.h`	C/C++ header file declaring functions that can be used to control the Palm OS Emulator.

Versions of Palm OS Emulator

Each released version of Palm OS Emulator has a version number that uses the following scheme:

```
<majorVers>.<minorVers>.<bugFix>[dab]<preRel>
```

Each field has the following semantics:

`majorVers` The major version number.

`minorVers` The minor version number.

`bugFix` The optional bug repair revision number.

`dab` The prelease stage of the product, as follows:

- `d` Indicates that the version is currently under development, and features are still being added.
- `a` Indicates alpha status, which means that the feature set is complete and some quality assurance testing has been performed.
- `b` Indicates beta status, which means that bugs uncovered in the alpha version have been addressed, and more extensive testing has been performed.

`preRel` The developmental, pre-release version number.

Some examples of version numbers are shown in Table 1.2

Table 1.2 Version number examples

Version	Description
2.0	Official release version 2.0
2.1d19	The 19th developmental release of version 2.1.
2.1a2	The 2nd alpha release of version 2.1.

Profile Versions

Some releases of Palm OS Emulator include a profile version, with the word profile appended to the program name. Each profile version adds the ability to perform selective profiling of your program's execution, and to save the results to a file.

The code required to add profiling capability slows down your application, even when you are not using profiling. That means that you are better off using the non-profiling version of Palm OS Emulator if you don't expect to use the profiling capabilities.

For more information about profiling with Palm OS Emulator, see Profiling Your Code.

Command Line Options

If you are running Palm OS Emulator on a Windows-based desktop computer or on a Unix system, you can supply the session parameters as command-line parameters. For example:

```
Emulator -psf C:\Data\Session1.psf
```

Table 1.3 shows the options that you can specify on the Windows command line. You can also change most of these options by starting a new session with the **New** menu command, as described in Configuring a New Session.

NOTE: The command line options are not available on Macintosh computers.

Note that the command line option specifications are not case sensitive.

Table 1.3 Palm OS Emulator command line options

Option syntax	Parameter values	Description
`-horde <num>`	A Gremlin number	The number of the Gremlin to run after the session is created or loaded. Note that this is equivalent to supplying the same Gremlin number for the `horde_first` and `horde_last` options.
`-horde_first <num>`	A Gremlin number	The first Gremlin to run in a horde.
`-horde_last <num>`	A Gremlin number	The last Gremlin to run in a horde.
`-horde_apps <app name list>`	A comma-separated list of applications	The list of applications to which the Gremlin horde is allowed to switch. The default is no restrictions.

Table 1.3 Palm OS Emulator command line options

Option syntax	Parameter values	Description
`-horde_save_dir <path>`	A path name	The name of the directory in which to save session and log files. The default log location is the directory in which the Palm OS Emulator application is stored.
`-horde_save_freq <num>`	An event count	The Gremlin snapshot frequency. The default value is to not save snapshots.
`-horde_depth_max <num>`	An event count	The maximum number of Gremlin events to generate for each Gremlin. The default value is no upper limit.
`-horde_depth_switch <num>`	An event count	The number of Gremlin events to generate before switching to another Gremlin in the horde. The default is to use the same value as specified for the `horde_depth_max` option.
`-psf <fileName>`	Any valid `.psf` file name	The emulator session file to load upon start-up. You can also load a session file with the **Open** menu command.
`-rom <fileName>`	Any valid ROM file name	The name of the ROM file to use.

Table 1.3 Palm OS Emulator command line options

Option syntax	Parameter values	Description
-ram <size> or -ramsize <size>	One of the following kilobyte size values: 128 256 512 1024 2048 4096 8192	The amount of RAM to emulate during the session.
-device <type>	One of the following device type values: Pilot PalmPilot PalmIII PalmIIIx PalmV PalmVx PalmVII PalmVIIEZ ColorDevice	The device type to emulate during the session. Note that Pilot1000 and Pilot5000 are synonyms for Pilot. Also note that PalmPilotPersonal and PalmPilotProfessional are synonyms for PalmPilot.
-load_apps <file name list>	A list of valid file names, separated by commas	A list of .prc or other files to load into the session after starting up.
-log_save_dir <path>	A path name	The name of the directory in which to save the standard log file. The default log location is the directory in which the Palm OS Emulator application is stored.

Table 1.3 Palm OS Emulator command line options

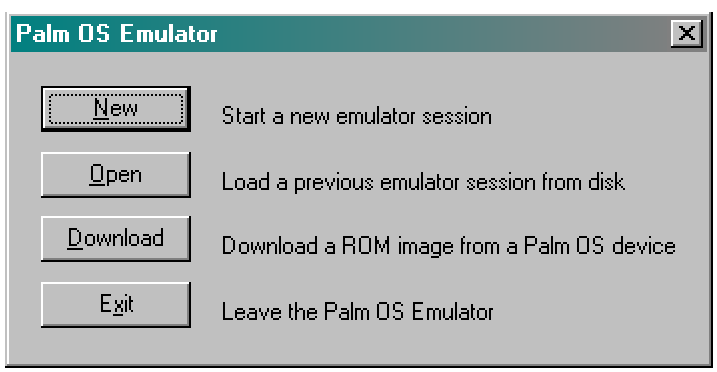
Option syntax	Parameter values	Description
`-quit_on_exit`	None	If the `-run_app` option was specified, this option indicates that Palm OS Emulator should quit after that application terminates.
`-run_app <app name>`	Application name	The name of an application to run in the session after starting up. You must specify the name of the application, not the name of the application's file.
`-silkscreen <type>` or `-skin <type>`	One of the following silkscreen types: `english` `japanese`	The silkscreen type to emulate during the session.

How Palm OS Emulator Starts Execution

When Palm OS Emulator starts execution, it determines its configuration by sequencing through the following rules:

1. If the *Caps Lock* key is on, the Startup dialog box is always displayed. The Startup dialog box is shown in Figure 1.2.

Figure 1.2 The Palm OS Emulator startup dialog box



NOTE: The dialog box shown in Figure 1.2 is displayed when you are running Palm OS Emulator on a Windows-based computer.

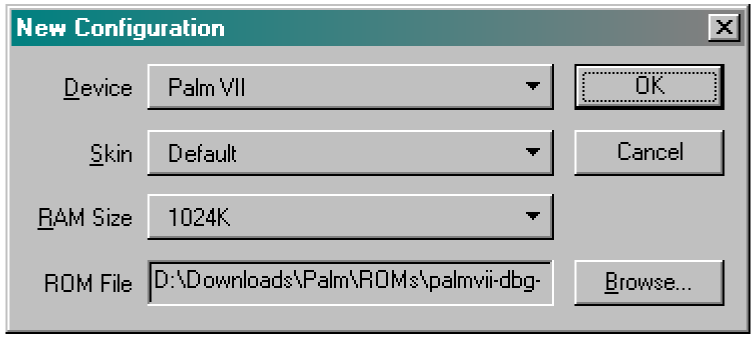
If you are using a Macintosh computer, the New Session dialog box, shown in Figure 1.6 is displayed instead.

If you are using a Unix system, Palm OS Emulator does not provide an automatic startup sequence; instead, it presents you with a window that displays a device graphic, and you must right-click in that window to display the new session menu.

2. If the Caps Lock key is not on, Palm OS Emulator scans the command line for options. If an error is encountered on the command line, Palm OS Emulator displays an error message and then presents the Startup dialog box.
3. If a session (`.psf`) file was specified on the command line, Palm OS Emulator attempts to load the file. If the file cannot be loaded, Palm OS Emulator displays an error message and then presents the Startup dialog box.
4. If any other options are specified on the command line, Palm OS Emulator attempts to start a new session with those values. If any of the four values is missing, Palm OS Emulator displays the session configuration dialog box, as shown in Figure 1.3.

   If any of the command line options are not valid, or if the user cancels the dialog box, Palm OS Emulator displays an error message and then presents the Startup dialog box.

Figure 1.3 The session configuration dialog box



5. If no command line options are specified, Palm OS Emulator attempts to reopen the session file from the most recent session, if one was saved. If the file cannot be opened, Palm OS Emulator displays an error message, and then presents the Startup dialog box.
6. Palm OS Emulator attempts to create a new session based on the setting most recently specified by the user. If an error occurs, Palm OS Emulator displays an error message, and then presents the Startup dialog box.

Probably the most common scenario is when you start Palm OS Emulator without any command line parameters, and it restarts with saved information from the previous session.

NOTE: When it starts up, Palm OS Emulator looks for the most recently saved .psf file. On Windows and Unix, the Emulator uses the full path name of that file; on Macintosh systems, the Emulator uses aliases to locate the file. If it cannot find that file, Palm OS Emulator looks for the file name in the directory in which the Palm OS Emulator executable is located.

The Palm OS Emulator User Interface

This section provides a description of the user interface for Palm OS Emulator, including descriptions of the menus and keyboard usage.

The Palm OS Emulator Display

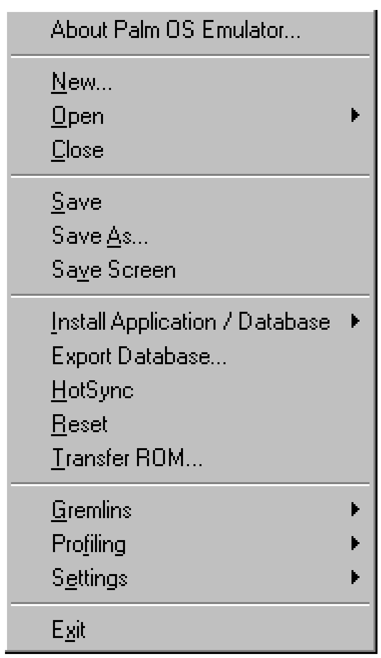
The Palm OS Emulator display looks very much like a real Palm Computing handheld device. You can use your mouse to perform actions that you perform with the stylus on handheld devices, and you can use the menus to access the Palm OS Emulator functionality.

Using the Menus

You can also access features that are specific to Palm OS Emulator by choosing menu commands:

- If you are using Windows, you right-click on the Palm OS Emulator screen display to access the menu items, or press the F10 key. The Palm OS Emulator menu displays, as shown in Figure 1.4.

Figure 1.4 The Windows version of the Palm OS Emulator menu



- If you are using a Macintosh, select menu commands from the menu bar. The Macintosh menu presents the same commands in four different menus, as described in Table 1.4. The Macintosh version is only slightly different:

- The Macintosh version of Palm OS Emulator uses the **Preferences** command instead of the **Properties** command to access the option-setting dialog box.
- The Macintosh version of the Emulator features the **Undo**, **Cut**, Copy, **Paste**, and **Clear** commands, which are not available in the Windows version.
- The Macintosh version of the Emulator uses the **Quit** command instead of the **Exit** command.
- The Macintosh version does not feature the **Breakpoints** command.

Table 1.4 Palm OS Emulator Macintosh menus

Menu	Commands
File	**New** **Open** **Close** **Save** **Save As** **Save Screen** **Install Application/Database** **HotSync** **Reset** **Transfer ROM** **Quit**
Edit	**Undo** **Cut** **Copy** **Paste** **Clear** **Preferences** **Logging Options** **Debug Options** **Skins**
Gremlins	**New** **Step** **Resume** **Stop**
Profile	**Start** **Stop** **Dump**

- If you are using Unix, Palm OS Emulator provides the same commands as are included with the Windows version, except that the **Breakpoints** command is not available. The Unix version of the menu pops up like the Windows version, and uses a different hierarchy, but presents the same commands.

Table 1.5 provides a brief description of the Palm OS Emulator menu commands, listed in alphabetical order.

Table 1.5 The Palm OS Emulator menu commands

Command	Description
Close	Closes and optionally saves the current emulator session.
Exit	Exits Palm OS Emulator. If you have unsaved changes in your session file, Palm OS Emulator optionally prompts you to save the file before exiting.
Gremlin:New	Create a new Gremlin and start running it.
Gremlin:Step	Step a Gremlin, after stopping.
Gremlin:Resume	Resume running of the Gremlin. NOTE: this command is only shown in Windows versions, and is not yet implemented.
Gremlin:Stop	Stop running the Gremlin.
Gremlin: Resume from control file	Resumes running of Gremlins from data that was previously saved in a file. For more information, see Using Gremlins.
HotSync	Allows you to synchronize the emulator session environment with the desktop computer. See Using the HotSync Application With the Palm OS Emulator for more information about the cabling requirements and other considerations for this command.

Table 1.5 The Palm OS Emulator menu commands

Command	Description
Install App/DB	Allows you to install an application into the emulator session, in the same way that a user would install it on the handheld with the Palm Install tool. For more information, see Installing Applications.
Export Database...	Exports a database to your desktop computer as as a `.pdb` or `.pqa` file, or exports an application to your desktop computer as a `.prc` file.
New	Displays the new configuration dialog box for initiating a new session.
Open	Displays the open file dialog box for opening a saved emulator session file.
Profiling:Start	Start profiling your application.
Profiling:Stop	Stop profiling your application.
Profiling:Dump	Save the profiling information to a file.
	For more information, see Profiling Your Code.
Reset	Resets the current emulation session, as if the reset button on the back of the handheld was pressed.
Save	Saves the current emulator session to an emulator `.psf` file.
Save As	Saves the current emulator session to an emulator `.psf` file.

Table 1.5 The Palm OS Emulator menu commands

Command	Description
Save Screen	Saves the current screen image as a bitmap file. **TIP:** The Save Screen command is a very convenient means of capturing screen images for documentation of Palm OS® applications.
Settings: Properties	Presents the properties dialog box, as described in Palm OS Emulator Properties.
Settings: Logging	Presents the logging options dialog box, as described in Logging Options.
Settings: Debug	Presents the debug options dialog box, as described in Debug Options.
Settings: Skins	Presents the skins dialog box, as described in Changing the Emulator's Appearance.
Settings: Breakpoints	Presents the breakpoints dialog box, as described in Setting Breakpoints.
Transfer ROM	Allows you to download a ROM image and save it to disk. You can then initiate a new session based on that ROM image. For more information, see Transferring a ROM Image From a Handheld.

Using the Hardware Buttons

Palm OS Emulator emulates each of the hardware buttons on Palm Computing devices. You can click on a button to activate it, and you can press and hold down a button just as you would on a handheld. Palm OS Emulator also allows you to activate the hardware buttons with keyboard equivalents, as shown in Table 1.6.

Table 1.6 Keyboard equivalents for the hardware buttons

Button	Keyboard equivalent
On/off	Esc
Palm Date Book	F1
Palm Address Book	F2
Palm To Do List	F3
Palm Memo Pad	F4
Up	Page Up
Down	Page Down

Entering Data

Palm OS Emulator allows you to use your desktop computer pointing device to tap and to draw Graffiti characters, just as you do with the stylus on the handheld.

Palm OS Emulator also allows you to enter text from the desktop computer keyboard. For example, you can type the text for a note by tapping in the note text entry area and then using the keyboard.

Control Keys

Palm OS Emulator also supports a set of control keys that you can use for input. These keys, which are shown in Table 1.7, are the same control keys that you can use with the Palm OS Simulator program.

Table 1.7 Palm OS Emulator Control Keys

Control key combination	Description
`Control - A`	Displays the menu
`Control - B`	Low battery warning
`Control - C`	Command character
`Contorl - D`	Confirmation character

Table 1.7 Palm OS Emulator Control Keys

Control key combination	Description
`Control - E`	Displays the application launcher
`Control - F`	Displays the onscreen keyboard
`Control - M`	Enters a linefeed character
`Control - N`	Jumps to the next field
`Control - S`	Automatic off character
`Control - T`	Sets or unsets hard contrasts
`Control - U`	Turns backlighting on or off

Loading ROM Images

Since the Palm OS Emulator emulates the Palm Computing Platform hardware, all components of the hardware must be present. This includes a ROM image file, which is not shipped with the Emulator. There are two ways to obtain a ROM image:

- download a ROM image from the Palm web site
- transfer a ROM image from a handheld

Downloading a ROM Image Obtained From Palm

To download a debug ROM image from Palm, go to the Palm developer zone web site, which is a rich source of resources for Palm OS developers. The developer zone URL is:

http://www.palm.com/devzone

The ROM image files are found in the Palm Provider Pavilion.

Transferring a ROM Image From a Handheld

To transfer a ROM image from a handheld, you need to follow these steps:

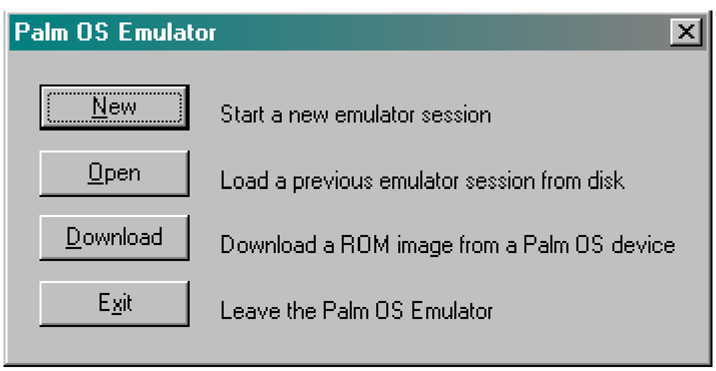
1. Install the Palm OS application named `ROM Transfer.prc` on your handheld device. You can use the Install program in the Palm Desktop organizer software and then synchronize with the handheld to install this program.
2. Place the handheld in the HotSync cradle that is connected to your desktop computer.
3. Follow the steps in the appropriate section below.

Transferring a ROM File in Windows

This section describes how to transfer a ROM image from a handheld on a Windows-based desktop computer. Before proceeding, you must have the `ROM Transfer.prc` program installed on the handheld, as described in the previous section.

If you are running the program for the first time, Palm OS Emulator presents the Startup dialog box shown in Figure 1.5. Click on **Download** to begin the transfer of a ROM image from a handheld.

Figure 1.5 The Palm OS Emulator startup dialog box



If you are not running Palm OS Emulator for the first time, it usually restarts the session that you most recently ran, as described in How Palm OS Emulator Starts Execution. To transfer a new ROM image for Palm OS Emulator to use, you can right-click on the Palm OS

Emulator display (the Palm device image) and select the **Transfer ROM** menu choice.

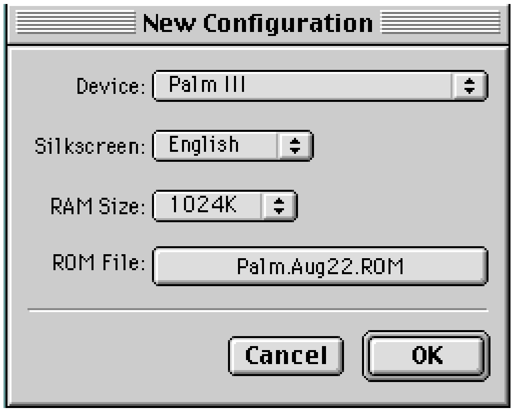
Once you have chosen to transfer a ROM image, Palm OS Emulator presents you with a sequence of dialog boxes that guide you through the process.

Transferring a ROM File On a Macintosh

This section describes how to transfer a ROM image from a handheld on a Macintosh desktop computer. Before proceeding, you must have the `ROM Transfer.prc` program installed on the handheld, as described in the previous section.

If you are running the program for the first time, Palm OS Emulator presents the dialog box shown in Figure 1.6.

Figure 1.6 Running Palm OS Emulator for the first time on a Macintosh system



You can dismiss this dialog box and choose the **Transfer ROM** command from the File menu.

If you are not running Palm OS Emulator for the first time, it usually restarts the session that you most recently ran. To transfer a new ROM image for Palm OS Emulator to use, select the **Transfer ROM** command from the File menu.

Once you have chosen to transfer a ROM image, Palm OS Emulator presents you with a sequence of dialog boxes that guide you through the process.

Using a ROM Image in Palm OS Emulator

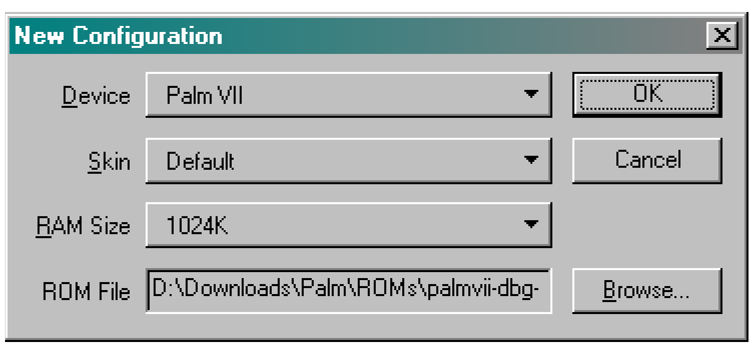
Once you have transferred a ROM image to disk, you need to create a new session that is based on the image. To initiate the new session, you select the **New** command. Table 1.8 shows the first step in creating a new session for each transfer method.

Table 1.8 Initiating a new session after transferring a ROM image

Method used to initiate ROM transfer	New session method
Clicked **Download** initial dialog in Windows	Click **New** in the dialog box.
Selected **Transfer ROM** command in Windows	Select either the **New** command or the **Close** command from the File menu.
Selected **Transfer ROM** menu command on a Macintosh	Select the **New** command from the File menu.

After you initiate the session, Palm OS Emulator presents the new configuration dialog box, which is described in Configuring a New Session. The Windows version of this dialog box is shown in Figure 1.7.

Figure 1.7 The New Configuration dialog box



After you select your parameters and click **OK**, Palm OS Emulator begins an emulation session.

Drag and Drop a ROM Image

You can use drag and drop to start a new Emulator session in either of two ways:

- Drag and drop a ROM image file onto the Emulator screen to start a new session.
- Drag and drop a ROM image file onto the Emulator executable or shortcut (alias) to start the Palm OS Emulator program.

You can also drag and drop other file types, as described in Dragging and Dropping Files.

Using the Binder to Create an Executable

If you are running the Palm OS Emulator on Windows NT, you can use the Binder program to create an executable that binds the Emulator program with a ROM image and optionally a RAM image. The bound program can then be used for demonstrations, training, and kiosk systems.

Testing and Debugging With Palm OS Emulator

This section provides an overview of testing and/or debugging an application with Palm OS Emulator.

Testing Software

Testing software is probably the most common use of Palm OS Emulator. This section provides a quick summary of the steps to load and test an application.

Debug Options

The Palm OS Emulator monitors the actions of your application while it is emulating the operation of the handheld device. When your application performs an action that does not strictly conform to Palm Computing's programming guidelines, the Emulator displays a dialog box that explains what is happening.

The debugging options dialog box, which is shown in Figure 1.8, allows you to enable or disable the monitoring activities applied to your application. Use the **Debug Options** command to display this dialog box.

Figure 1.8 The Palm OS Emulator debugging options dialog box

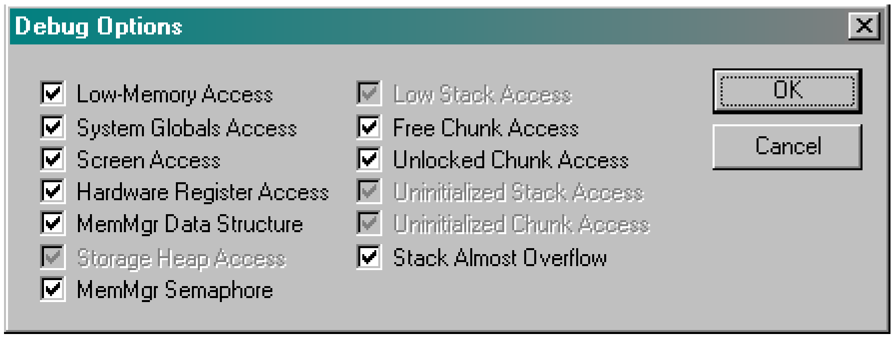


Table 1.9 describes each of the debugging options.

Table 1.9 Emulator debugging options

Option	Description
Low-Memory Access	Monitors low-memory access by applications. Low-memory access means an attempt to read from or write to a memory location in the range `0x0000` to `0x00FF`.
System Globals Access	Monitors access to system global variables by applications. System global variable access is defined as reading from or writing to a memorylocation in the range from `0x0100` to the end of the trap dispatch table.

Table 1.9 Emulator debugging options *(continued)*

Option	Description
Screen Access	Monitors LCD screen buffer access by applications. LCD screen buffer access is defined as reading from or writing to the memory range indicated by the LCD-related hardware registers.
Hardware Register Access	Monitors accesses to hardware registers by applications. Hardware register access is defined as reading from or writing to memory in the range from `0xFFFFF000` to `0xFFFFFFFF`.
MemMgr Data Structure	Monitors access to Memory Manager data structures, which is restricted to only the Memory Manager. Memory Manager data structures are the heap headers, master pointer tables, memory chunk headers, and memory chunk trailers.
Storage Heap Access	Monitors naked access to the storage heap by applications. To access the storage heap, your application should use the `DmWrite` functions.
MemMgr Semaphore	Monitors how long the Memory Manager semaphore has been acquired for write access using the `MemSemaphoreReserve` and `MemSemaphoreRelease` functions. Your applications should not be calling these functions; however, if you must call them, you should not hold the semaphore for longer than 10 milliseconds.
Low Stack Access	Monitors access to the range of memory below the stack pointer.

Table 1.9 Emulator debugging options *(continued)*

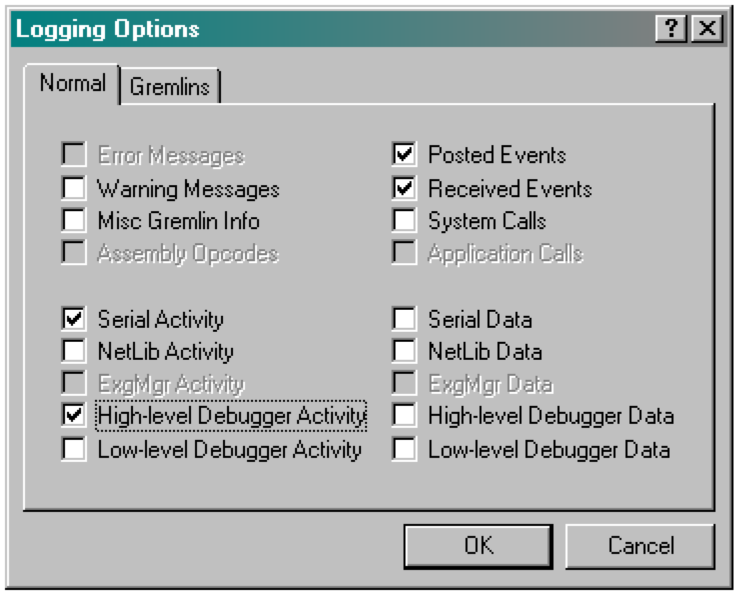
Option	Description
Free Chunk Access	Monitors access to free memory chunks. No process should ever access the contents of a chunk that has been deallocated by the `MemChunkFree`, `MemPtrFree`, or `MemHandleFree` functions.
Unlocked Chunk Access	Monitors access to unlocked, relocatable memory chunks, which is restricted to the Memory Manager.
Uninitialized Stack Access	Monitors read accesses to uninitialized portions of the stack. You can use this option to detect read accesses to uninitialized local variables.
Uninitialized Chunk Access	Monitors read access to uninitialized portions of memory chunks that have been allocated by the `MemChunkNew`, `MemPtrNew`, and `MemHandleNew` functions. You can use this option to detect read accesses to uninitialized portions of dynamically allocated memory chunks. Note that your application's global variables are stored in memory chunks allocated by these functions, so enabling this option also detects read accesses to uninitialized global variables.
Stack Almost Overflow	Ensures that the stack pointer has not dipped below the space allocated for it by the kernel. When this option is enabled, Palm OS Emulators warns you when the application stack is getting close to full. Note that you are always warned of a stack overflow, even if this option is disabled.

Logging Options

The Palm OS Emulator also logs various actions taken by your application to help you debug and performance tune your code. The logged information is automatically written to a text file that is saved in the same directory as the Emulator executable.

You can control the logging activity with the logging options dialog box, which is shown in Figure 1.9. Use the **Logging Options** command to display this dialog box.

Figure 1.9 Palm OS Emulator logging options



The logging options dialog box features separate tabs for logging during normal operations, and for logging while a Gremlin is running. Both tabs offer the same options, which are described in Table 1.10

Table 1.10 Emulator logging options

Option	Description
Error Messages	Not yet implemented.
Warning Messages	Logs any message that is displayed in a dialog box that can be dismissed by tapping the Continue button.
Misc Gremlin Info	Logs information about Gremlins that is mostly useful for debugging the Gremlins themselves.

Table 1.10 Emulator logging options *(continued)*

Option	Description
Assembly Opcodes	Logs assembly-level trace information, including registers, the program counter, opcodes, and related information. This option is not yet implemented.
Posted Events	Logs events that have entered into the system by way of calls to the `EvtAddEventToQueue`, `EvtAddUniqueEventToQueue`, `EvtEnqueuePenPoint`, and `EvtEnqueueKey` functions.
Received Events	Logs events returned by calls to the `EvtGetEvent`, `EvtGetPen`, and `EvtGetSysEvent` functions.
System Calls	Logs calls to Palm OS® functions.
Application Calls	Logs calls to functions in your application. This option is not yet implemented.
Serial Activity	Logs changes in serial port settings, and the opening and closing of the serial port.

Table 1.10 Emulator logging options *(continued)*

Option	Description
Serial Data	Logs data sent and received over the serial port. Data is logged as it is being transferred over the host serial port Incoming data follows this path: 1. Serial port 2. Emulated hardware registers 3. Palm OS 4. Palm application Palm OS Emulator logs the serial port data. Outgoing data follows this path: 1. Palm application 2. Palm OS 3. Emulated hardware registers 4. Serial port Again, Palm OS Emulator logs the serial port data.
NetLib Activity	Logs calls to `NetLib` functions, including parameter and return values.
NetLib Data	Logs data sent and received via `NetLib` calls.
ExgMgr Activity	Not yet implemented.
ExgMgr Data	Not yet implemented.
High-level Debugger Activity	Logs messages received back from an external debugger, and the messages sent back to the debugger.
High-level Debugger Data	Logs details of the messages sent to and received from an external debugger.

Table 1.10 Emulator logging options *(continued)*

Option	Description
Low-level Debugger Activity	Traces the low-level mechanisms that receive raw data from external debuggers and send data back to external debuggers.
Low-level Debugger Data	Logs the raw data being sent to and received from an external debugger.

Using Gremlins

You can use Gremlins to automate testing of an application. A **Gremlin** generates a series of user input events that test your your application's capabilities. You can have a Gremlin to run a specified number of times, or to loop forever, which allows you to set up a Gremlin and allow it to run overnight to thoroughly test your application.

A **Gremlin horde** is a range of Gremlins that you want Palm OS Emulator to run. The Emulator generates a stream of events for each Gremlin and then moves onto the next Gremlin. The Emulator cycles through the Gremlins until the maximum number of events have been generated for the horde.

The Palm OS Emulator generates a stream of events for each Gremlin in the horde until one of the following conditions occurs:

- An error such as a hardware exception or illegal memory access is generated.
- The maximum number of events for a single Gremlin have been generated.
- The maximum number of events for the horde have been generated.
- You stop the horde by choosing the Stop or Step command from the Emulator menu or from the Gremlin Status dialog box.

If a Gremlin generates an error, it is halted and the Palm OS Emulator does not include it when cycling through the horde again.

Gremlin Characteristics

Each Gremlin has the following characteristics:

- it generates a unique, random sequence of stylus and key input events to step through the user interface possibilities of an application
- it has a unique “seed” value between 0 and 999
- it generates the same sequence of random events whenever it is run
- it runs with a specific application or applications
- it displays a report immediately when an error occurs

Gremlin Horde Characteristics

Each Gremlin horde has the following characteristics:

- The number of the first Gremlin to run. This must be a value between 0 and 999.
- The number of the last Gremlin to run. This must be a value between 0 and 999.
- The switching depth of the Gremlin horde. This is the number of events to run for each Gremlin. After this many events have been generated for the Gremlin, it is suspended, and the next Gremlin in the horde starts running.
- The maximum number of events for each gremlin in the horde. The Emulator stops running the Gremlin after it posts this many events, or after it terminates with an error.
- With which applications the Gremlins are to run. You can select a single application, a group of applications, or all applications.
- Errors that occur are logged to the log file and the emulation continues with the next Gremlin in the horde.

When Palm OS Emulator runs a Gremlin horde, it actually maintains a separate stream for each Gremlin in the horde. When it starts a horde, the Emulator first saves the complete state of the emulation to a session (`.psf`) file. Then, the Emulator:

- Starts the first Gremlin. When the Gremlin has posted a number of events equal to the specified switching depth, the

Emulator saves its state to a new file and suspends the Gremlin.

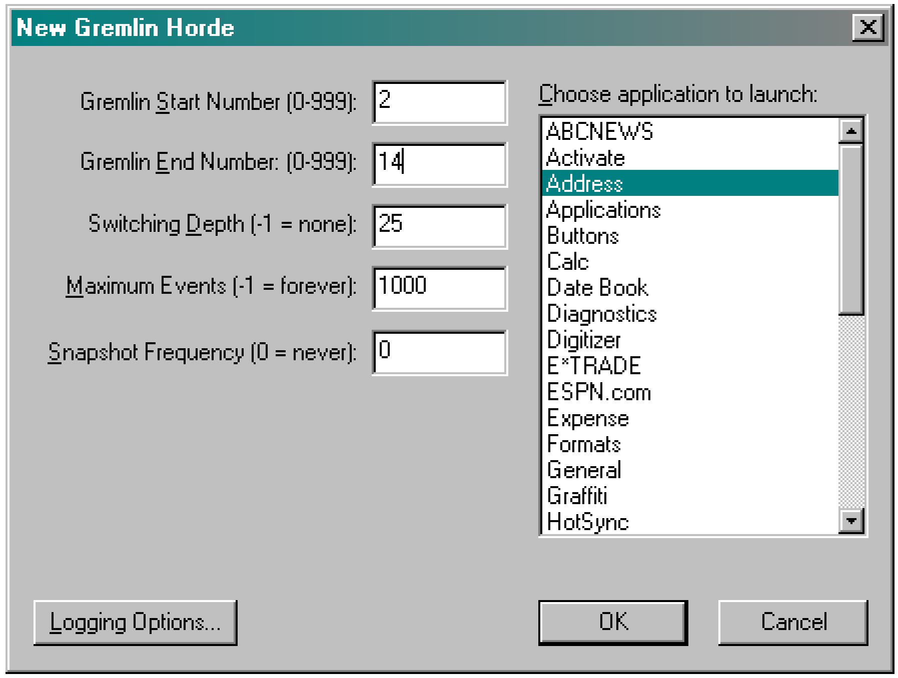
- Reloads the original session state.
- Starts the second Gremlin andruns it until it posts that number of events, at which time its state is saved to another file, and the Gremlin is suspended.
- Runs each Gremlin in the horde, until each has been suspended or terminated:
  - A Gremlin is terminated when an error occurs while the Gremlin is posting events.
  - A Gremlin is suspended when it has posted a number of events equal to the switching depth for the horde.
- Returns to the first suspended Gremlin in the horde, reloads its state from the saved file, and resumes its execution as if nothing else had happened in the meantime.
- Continues cycling through the Gremlins in the horde until each Gremlin has finished. A Gremlin finishes when either of these conditions occurs:
  - the Gremlin has terminated due to an error
  - the Gremlin has posted a total number of events equal to the maximum specified for the horde.

Running a Gremlin Horde

Select the **New Gremlin** command to start a Gremlin. The new Gremlin dialog box displays, as shown in Figure 1.10. You use this dialog box to specify the characteristics of the Gremlin horde that you want to run.

NOTE: If you wish to run a single Gremlin, simply set the Gremlin Start Number and Gremlin End Number fields to the same value.

Figure 1.10 The Gremlin horde dialog box



When Palm OS Emulator runs the example shown in Figure 1.10, the horde will operate as follows:

- The Emulator will only run the Address application when generating key and stylus events for this horde.
- The Emulator will use a seed value of 2 for the first Gremlin in the horde and a seed value of 14 for the last Gremlin in the horde. It also runs all intervening Gremlins: numbers 3 through 13.
- The Emulator will generate 25 events for each Gremlin before switching to the next Gremlin in the horde.
- The Emulator will cycle through the Gremlins in the horde until a total of 1000 events have been generated for each Gremlin. Thus, a total of 13,000 events will be generated.

This means that the Emulator will generate the following sequence of Gremlin events:

1. Gremlin #2 runs and receives twenty-five events, after which Gremlin 2 is suspended.

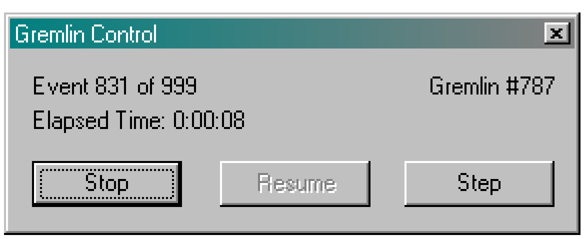
2. Gremlin #3 runs and receives twenty-five events, after which Gremlin #3 is suspended.
3. Similarly, each Gremlin (#4 through #14) runs and receives twenty-five events, after which it is suspended.
4. The Emulator loops back to Gremlin #2 and runs it, sending it twenty-five events before again suspending it.
5. Gremlin #3 runs again, receives twenty-five events, and suspends.
6. This looping through the Gremlins and sending each events until the switch depth (25) is reached continues until the maximum number of horde events (1000) have been generated.
7. All activity for the Gremlin horde completes.

NOTE: If an error occurs while a specific Gremlin is running, Palm OS Emulator halts that Gremlin rather than suspending it. This means that the Gremlin is not run when the Emulator next iterates through the horde.

Stepping and Stopping Gremlins

After the horde starts running, Palm OS Emulator displays the Gremlin control dialog box, which is shown in Figure 1.11. You can use the commands in this dialog box to stop, resume, and single-step a Gremlin. You can also use the **Gremlins** menu command to perform these actions.

Figure 1.11 The Gremlin status dialog box



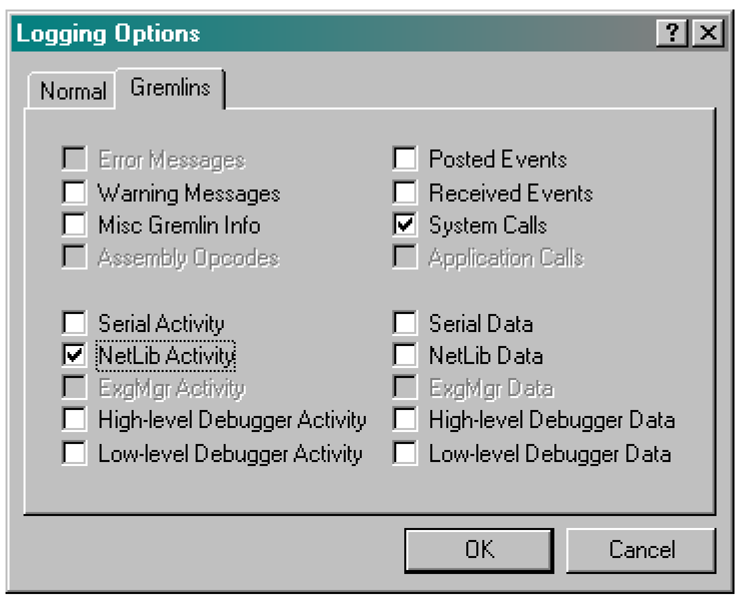
Gremlin Snapshots

When you start a new Gremlin horde, you can specify that you want the Palm OS Emulator to take a snapshot on a regular basis. You specify a frequency value, as shown in Figure 1.10, and the Emulator saves a session file each time that many Gremlins have run. Each snapshot is a `.psf` file that captures the current state of the emulation. You can open the snapshot in the Emulator as a new session and begin debugging from that state.

Logging While Gremlins Are Running

Palm OS Emulator allows you to specify separate logging options to use while Gremlins are running. Figure 1.12 shows the Gremlin logging options dialog box. Each of the options is described in Logging Options.

Figure 1.12 Gremlin logging options



Setting Breakpoints

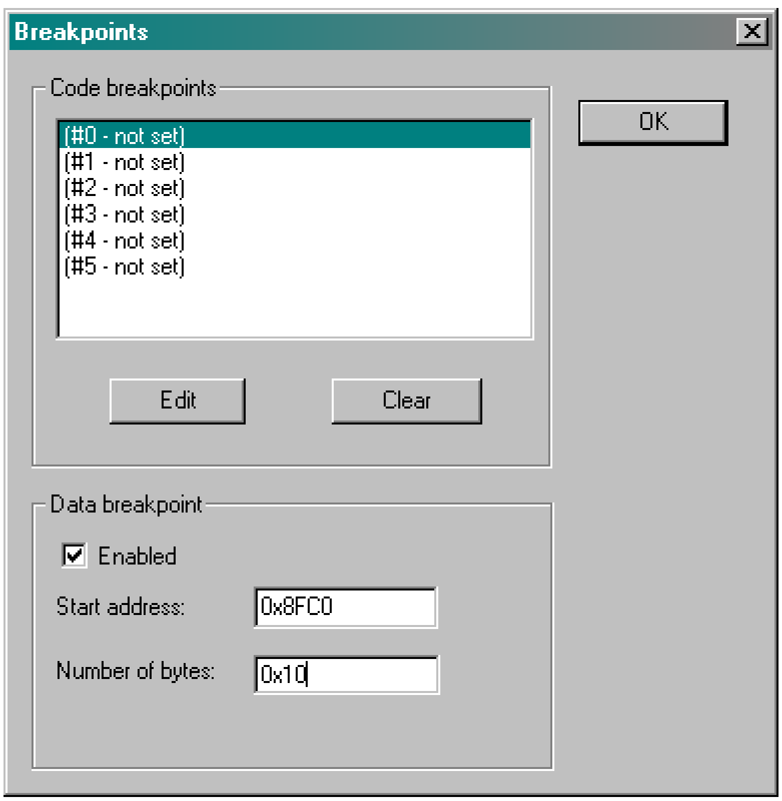
You can set breakpoints in your code with the Emulator. When the Palm OS Emulator encounters a breakpoint that you have set, it halts and takes one of the following actions:

- If you are running the Emulator connected to a debugger, the Emulator sends a message to the debugger, informing it that the breakpoint was hit. The debugger then handles that command as it sees fit.
- If the Emulator is not connected to a debugger, the Emulator displays an error message. This message will typically say something like “TRAP $0 encountered.”

To set a breakpoint, select the **Breakpoints** command from the **Settings** menu. The Breakpoints dialog box is displayed, as shown in .

NOTE: You cannot use the Breakpoints feature on the Macintosh or Unix versions of the Palm OS Emulator.

Figure 1.13 Setting a breakpoint



Setting the Data Breakpoint

You can set exactly one data breakpoint. While your program is executing, the Emulator watches the specified address range; if it is written to, the Emulator generates a break. You can specify both the address and number of bytes fields in either hexadecimal (`0x`) or decimal.

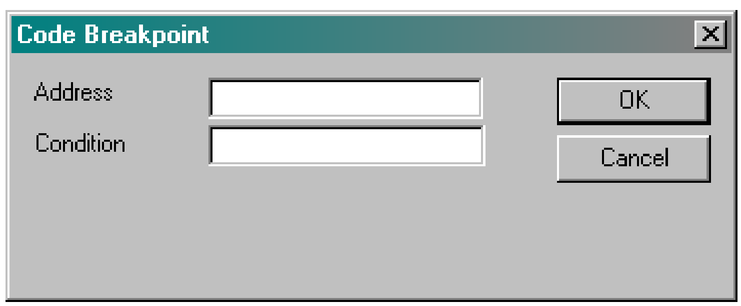
Setting Conditional Breakpoints

You can set up to six independent conditional breakpoints. The Emulator generates a break for a conditional breakpoint when both of the following are true:

- the program counter reaches the specifies address
- the specified condition is true

To set one of these breakpoints, select the breakpoint number in the list at the top of the dialog box, and click on the **Edit** button. This displays the Code Breakpoint dialog box, which is shown in Figure 1.14.

Figure 1.14 Setting a code breakpoint



To set the breakpoint, specify an address and the break condition. You can specify the address in hexadecimal (0x) or decimal.

The condition that you specify must have the following format:

```
<register> <condition> <constant>
```

`register` One of the registers: `A0`, `A1`, `A2`, `A3`, `A4`, `A5`, `A6`, `A7`, `D0`, `D1`, `D2`, `D3`, `D4`, `D5`, `D6`, or `D7`.

`condition` One of the following operators: ==, !=, <, >, <=, or >=.

`constant`	A decimal or hexadecimal constant value.

WARNING! All comparisons are unsigned.

Source Level Debugging

Palm OS Emulator provides an interface that external debugger applications can use to debug an application. For example, Metrowerks has developed a plug-in module that you can use to debug an application that Palm OS Emulator is running, in exactly the same manner as you would debug an application running on the handheld. This plug-in module is shipped with the latest version of CodeWarrior for Palm OS.

Connecting the Emulator With Palm Debugger

You can use the Palm Debugger with the Palm OS Emulator to perform extensive debugging of your applications. To use Palm Debugger with the Emulator, follow these steps:

1. Start the Palm Debugger and Palm OS Emulator programs.
2. In the Palm Debugger Communcations menu, select **Emulator**. This establishes the emulator program as the "device" with which Palm Debugger is communicating.
3. In the debugger window, type the `att` command.

You can now send commands from the Palm Debugger to the Palm OS Emulator.

Connecting the Emulator With External Debuggers

Palm OS Emulator can communicate with external debuggers using the methods shown in Figure 1.11.

Table 1.11 Palm OS Emulator Connections

Connection type	Platforms
TCP	All
PPC Toolbox	Macintosh
Memory-mapped files	Windows

NOTE: Currently, PalmDebugger uses TCP only when running on Windows. The CodeWarrior plug-in uses TCP if you select the `Use sockets` checkbox in the debugger preference panel.

However, although you can configure the TCP port that Palm OS Emulator uses, you cannot configure which TCP port that either PalmDebugger or the CodeWarrior plug-in uses.

If you are communicating with a debugger using TCP, you can configure which socket port the debugger connects to by editing the value of the `DebuggerSocketPort` preference setting in your preferences file. You can disable the TCP connection by setting the value of the `DebuggerSocketPort` preference to `0`.

NOTE: In some versions of Palm OS Emulator, you may notice that an unwanted PPP dial-up starts whenever you begin a new emulation session. You can disable this behavior by disabling the use of TCP for commmunications, which you do by setting the `DebuggerSocketPort` preference to `0`.

Profiling Your Code

One of the features of the Palm OS Emulator that is most useful for developers is the ability to profile your application while it is running, and to save the resulting data to a file that you can examine.

When the Emulator profiles your application, it monitors and generates statistics about where your code is spending its time, which enables you to focus your optimization efforts in the most productive manner.

You can start a profiling session by choosing the **Profiling Start** command. While profiling is active, the Palm OS Emulator monitors which application and system functions are executed, and the amount of time executing each. The Emulator collects the timing information until you select the **Profiling Stop** command.

You can then save the profiling information to a file by selecting the **Profiling Dump** command. The information is saved to file in two different formats. Both of these files are stored in the directory in which the Emulator executable is located:

File name	Description
`Profile Results.txt`	A text version of the profiling results.
`Profile Results.mwp`	A Metrowerks Profiler version of the results, which can be used with the MW Profiler application bundled with CodeWarrior Pro. **IMPORTANT:** The MW Profiler is only available on Macintosh computers.

You do not have to prepare your code in any special way for Palm OS Emulator to profile it, because the Emulator can determine when functions are entered and exited on its own. And the Emulator performs its profiling calculations between cycles, thus the timing information is quite accurate.

NOTE: It is a good idea to set your compiler's switch to embed debug symbols in your code so that you can easily interpret the profiling results.

Palm OS Emulator Session Features

Palm OS Emulator uses the concept of an emulation session, which is a testing or debugging session for a combination of the following items:

- the handheld device type to emulate
- the amount of RAM to emulate
- the ROM file to use for the emulation

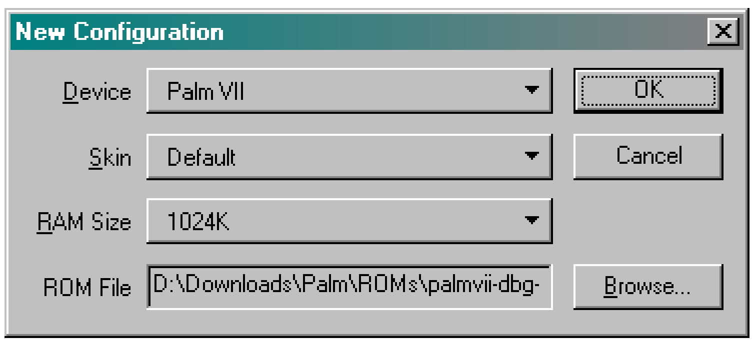
You can start new emulation sessions during a single run of Palm OS Emulator. You can also save the current state of a session and restore it in a later session. This session describes these features of Palm OS Emulator.

Configuring a New Session

You can start a new session in Palm OS Emulator by choosing the **New** command from the Palm OS Emulator menu. If you are already running an emulation session, Palm OS Emulator will optionally ask if you want to save the session in a Palm OS Emulator session (`.psf`) file before starting the new session. You set this option in your preferences.

Figure 1.15 shows the New Configuration dialog box, which Palm OS Emulator displays when you choose the **New** command from the menu.

Figure 1.15 Configuring a new session



You need to make the following choices in this dialog box:

- Select the Palm handheld device that you want to emulate in the session. You can choose from among the following choices:
  - Pilot (1000 / 5000)
  - PalmPilot (Personal / Pro)
  - Palm III
  - Palm IIIx
  - Palm V

- Palm VII
- Palm VII EZ
- Color Device

- Select the silkscreen that you want displayed on the emulation screen. Alternative silkscreens, such as the Japanese silkscreen, are only available for certain device types. The Default choice is always available, even when alternatives are not available.
- Select the amount of memory that you want emulated. You can choose from the following RAM sizes:
  - 128K
  - 256K
  - 512K
  - 1024K
  - 2048K
  - 4096K
  - 8192K

  Note that 1 MB is most often the right amount of RAM to emulate.
- Select the ROM file on your desktop computer that you want to use for the session. You can use the **Browse** button to navigate to the file. For more information about ROM files, see Loading ROM Images.

After you click the **OK** button, Palm OS Emulator begins an emulation session.

Dragging and Dropping Files

You can drag and drop the following file type categories onto the Palm OS Emulator LCD screen:

- `.prc`, `.pdb`, and `.pqa` files
- `.rom` files
- `.psf` files

When dragging and dropping files, you must observe the following rules:

- You can drag and drop only one `.rom` file at a time.
- You can drag and drop only one `.psf` file at a time.
- You can drag and drop any number of `.prc`, `.prb`, and `.pqa` files.
- You cannot drag and drop files from more than one of the file type categories in the same operation.

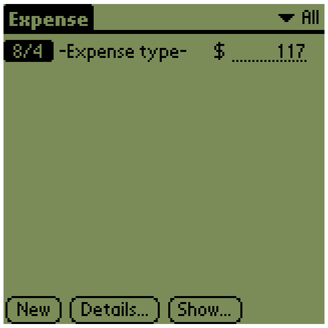
Saving and Restoring Session State

You can save the current state of a Palm OS Emulator session to a session file for subsequent restoration. Palm OS Emulator saves a session to a session file. The Emulator uses Save and Save As in the standard manner, with one addition: you can automate what happens when closing a session by changing the Save options.

Saving the Screen

You can save the current screen to a bitmap file by selecting the **Save Screen** menu command, which saves the contents of the emulated Palm handheld device screen.

Figure 1.16 A Palm OS Emulator screen shot

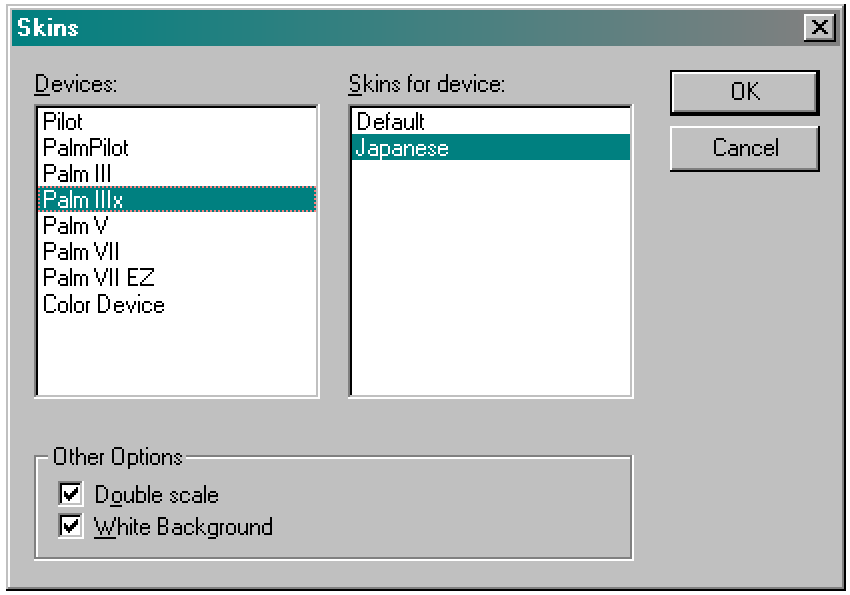


Palm OS Emulator saves screen images on Windows-based systems as `.bmp` bitmap images, and saves screen images on MacOS-based systems as SimpleText image files.

Changing the Emulator's Appearance

You can change the appearance of the Palm OS Emulator by choosing the **Skins** command from the Settings submenu. This displays the Skins dialog box, which is shown in Figure 1.17.

Figure 1.17 Changing the Palm OS Emulator appearance



The Skins dialog box provides three appearance options that you can use:

- Select or deselect the **Double scale** option to display the emulated device in double size or actual size on your monitor.
- Select or deselect the **White Background** option to display the emulated device LCD background color in white or green on your monitor.

NOTE: The term "skin" is used to refer to a set of graphics that an application uses to creates its appearance. You can change the appearance of an application by changing its skin.

The Palm OS Emulator Runtime Environment

This section describes how you can modify the Palm OS Emulator runtime environment, including changing the properties and installing applications in the emulator session.

Palm OS Emulator Properties

You can use the Properties dialog box to modify characteristics of your Palm OS Emulator sessions. To display this dialog box, choose the **Properties** menu command on a Windows system, or the **Preferences** menu command on a Macintosh system. The Properties dialog box is shown in Figure 1.18.

Figure 1.18 Changing the Palm OS Emulator properties

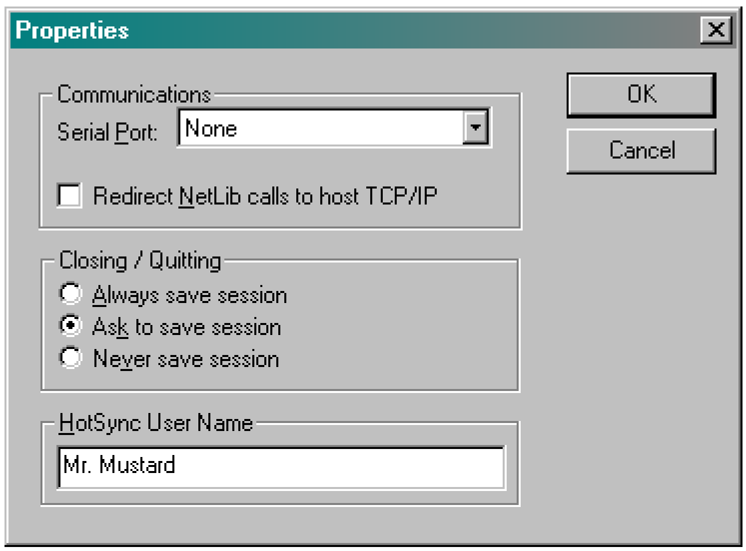


Table 1.12 describes the options available in the properties dialog box.

Table 1.12 Palm OS Emulator properties

Option	Description
Serial Port	Specifies which serial port the Palm OS Emulator uses to emulate serial communications on the handheld device.
Redirect Netlib calls	Redirects Netlib calls in emulated software to TCP/IP calls on the desktop computer.
Session saving	Selects what action the Palm OS Emulator takes when you close a session or quit the program.
User name	Selects the user account name for synchronizing from Palm OS Emulator with the desktop computer HotSync application.

Preferences Files

Your properties are stored in a preferences file on your computer. Each property is stored as a text string that you can view with a text editor. The location of your preferences file depends on the type of computer that you are using, as shown in Table 1.13.

Table 1.13 Palm OS Emulator preference file locations

Platform	File name	File location
Macintosh	`Palm OS Emulator Preferences`	In the `Preferences` folder.
Windows	`Palm OS Emulator Preferences.ini`	In the `Windows System` directory.
Unix	`.poserrc`	In your home directory.

Installing Applications

You can use the **Install** command to load applications or databases directly into the current Palm OS Emulator emulation session.

- in Windows, right-click on the Palm OS Emulator screen display and choose the Install Application / Database command
- on a Macintosh, select the Install Application / Database command from the File menu

The **Install** command displays an open file dialog box in which you can choose the application (`.prc`), database (`.pdb`), or Palm Query Appplication (`.pqa`) file that you want installed.

Palm OS Emulator immediately loads the file into emulated RAM. If Palm OS Emulator finds another application or database with the same creator ID, that application or database is deleted before the new version is loaded.

WARNING! If you install an application while the Palm OS Launcher is running, the Launcher does not update its data structures, and thus does not reflect the fact that a database has been added or modified. It is best to use the Install command while an application is running in the emulated session.

Serial Communications and Palm OS Emulator

The Palm OS Emulator supports emulation of the Palm device serial port connection. It does so by mapping Palm OS serial port operations to a communications port on the desktop computer. To select which port the Emulator uses, use the **Properties** (on Macintosh computers, this is **Preferences**) menu command, as described in Palm OS Emulator Properties.

When emulated software accesses the Dragonball or Dragonball EZ serial port hardware registers, Palm OS Emulator performs the appropriate actions on the specified serial port on the desktop computer. This means that serial read and write operations work as follows:

- when outgoing data is written to the UART's `tx` register, the Emulator redirects that data to the desktop computer's serial port.
- when the emulated software attempts to read data from the UART's `rx` register, the Emulator reads data from the desktop computer's serial port and places the data into that register.

Using the HotSync Application With the Palm OS Emulator

You can perform a HotSync operation from your emulated session in one of two ways:

- If you are using a Windows-based computer, you can use the Network HotSync option, which greatly simplifies your communications efforts.
- If you are not using a Windows-based computer, or your computer is not connected to a network, you can use a null-modem cable to connect two ports together and perform a HotSync operation.

Synchronizing From Palm OS Emulator With a Network

To synchronize when you are connected to a network, you need to set up your HotSync Manager application to perform a network synchronization. You do not need to use a null-modem cable when performing a network synchronization with the Palm OS Emulator.

Synchronizing From Palm OS Emulator Without a Network

To synchronize when you are not connected to a network, you need to connect the serial port that the HotSync application uses to communicate with the handheld device to another serial port that the Palm OS Emulator uses. You connect these ports together with a null modem cable, such as a LapLink cable.

For example, if your are using a Windows-based computer and your HotSync application uses the COM1 port, follow these steps:

1. Select the **Properties** (**Preferences** on a Macintosh) command and specify the COM2 port for use the Palm OS Emulator.

2. Connect COM1 and COM2 together with a null modem cable.
3. Select the HotSync command from the Palm OS Emulator menu.

The HotSync application synchronizes with the Palm OS Emulator just as it does with an actual hardware handheld device.

TIP: The desktop HotSync application is CPU intensive, which is not generally an issue; however, when you are using the HotSync application with the Palm OS Emulator, the two programs are sharing the same CPU, which can dramatically the synchronization down.

A handy trick to deal with this problem is to click on the Palm OS Emulator window after the HotSync process starts. This brings the Emulator back into the foreground and allows it to use more CPU time, which improves the speed of the overall process.

If your desktop computer has two ports and you use a serial mouse on one of them, you can temporarily disable the mouse, perform a synchronization, and reenable the mouse. Follow these steps:

1. Disable your mouse.
2. Restart Windows.
3. Connect the serial ports together with a null modem cable.
4. Start the Palm OS Emulator.
5. Press F10 to display the menu, then H to begin the HotSync operation.
6. After the HotSync operation completes, reenable your mouse.
7. Restart Windows again.

TIP: When you first perform a HotSync operation with the Palm OS Emulator, the HotSync application asks you to select a user name. It is a good idea to create a new user account, with a different name, for use with the Emulator.

Palm OS Emulator Error Handling

This section describes the error handling and reporting features of the Palm OS Emulator program, including the following information:

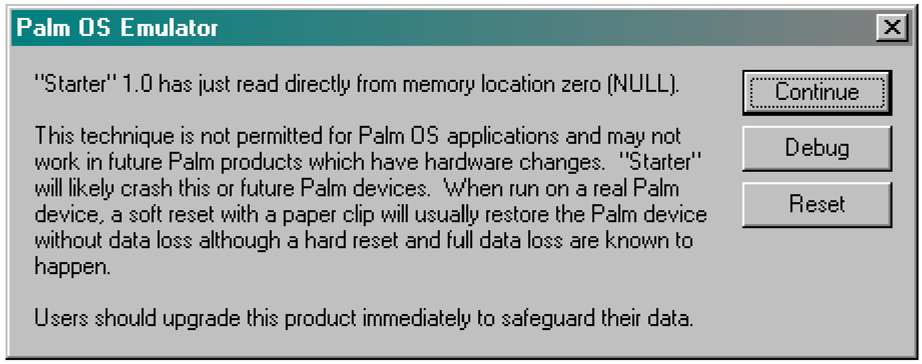
- which conditions are detected
- what the Emulator does upon detecting an error condition
- the message displayed for each error condition
- the options available to the user when an error condition occurs

Detecting an Error Condition

When Palm OS Emulator detects an error condition, it generates error message text and displays the error dialog box. If you select the Debug button in the error dialog box, the Emulator attempts to send the text to an external debugger such as Palm Debugger or MWDebug; if successful, the Emulator then stops emulating opcodes until the external debugger sends a command specifying that it can resume emulation.

If the Emulator cannot send the text to a debugger, it presents the error text to the user in a dialog box like the one shown in Figure 1.19.

Figure 1.19 Palm OS Emulator error dialog box



You can click one of the three buttons in the dialog box:

Button	Description
Continue	Continues emulation, if possible.
Debug	Enters the external debugger, if one is running.
Reset	Performs a soft reset on the emulated device ROM.

Error Condition Types

The Palm OS Emulator detects condition types:

- A *processor exception* involves the CPU pushing the current program counter and processor state onto the stack, and then branching through a low-memory vector.
- A *memory access exception* involves access to a memory location that the application is not supposed to access.
- An *application error message* is a message displayed when software running on the handheld device calls a system function such as `ErrDisplayFileLineMsg` or `SysFatalAlert`.

The Palm OS Emulator uses four levels of accessibility when checking memory accesses:

- Applications have the least access to memory. An application is any software running in RAM on the handheld device.

- The system has more access to memory than do applications. The system is any software running in ROM on the handheld device.
- The memory manager has the most access to memory. The memory manager is any function operating within the context of a memory manager call, which means any function that runs while a memory manager function is still active.
- Some sections of memory cannot be accessed by any software.

Error Messages

Table 1.14 shows the Palm OS Emulator error messages. Note that you can prevent some of these messages by disabling the relevant debugging option, as described in Debug Options.

Table 1.14 Palm OS Emulator error messages

Error type	Description	Message example
Hardware register access	The application or system software has accessed a Dragonball or Dragonball EZ hardware register.	`"Mytest" 1.0 has just read directly from the hardware registers.`

Low-memory access	The application or system software has accessed low memory (the first 256 bytes), which contains the exception vectors.	`"Mytest" 1.0 has just read directly from low memory.`
		or
		`"Mytest" 1.0 has just read directly from NULL (memory location zero)`

System variable access	The application or system software has accessed a system variable, which resides in a memory location between low memory and the the end of the system function dispatch table.	`"Mytest" 1.0 has just read directly from Palm OS global variables.`

Table 1.14 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
LCD screen buffer access	The application or system software has accessed the screen buffer, which is defined by the LCD-related hardware registers.	`"Mytest" 1.0 has just read directly from screen memory.`
Memory Manager data structure access	The application or system software has accessed a memory manager data structure, which includes heap headers, master pointer tables, chunk headers, and chunk trailers.	`"Mytest" 1.0 has just read directly from memory manager data structures.`
Unlocked chunk access	The application or system software has accessed an unlocked memory chunk.	`"Mytest" 1.0 has just read directly from an unlocked memory chunk.`
Low-stack access	The application or system software has accessed an area of the stack below the stack pointer. The stack is defined by values returned by the `SysGetAppInfo` function when it is called during system startup. If Palm OS Emulator cannot determine the stack range, it does not monitor low-stack accesses.	`"Mytest" 1.0 has just read directly from an invalid section of memory known as the "stack" .`

Table 1.14 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
Uninitialized stack access	The application or system software has accessed uninitialized memory, which is memory that has not previously been written.	`"Mytest" 1.0 has just read directly from an uninitialized section of memory known as the "stack" .`
Free chunk access	The application or system software has accessed an unallocated memory chunk.	`"Mytest" 1.0 has just read directly from an unallocated chunk of memory.`
Uninitialized chunk access	The application or system software has attempted read access to uninitialized memory.	`"Mytest" 1.0 has just read directly from an uninitialized chunk of memory.`
Storage heap access	The application has accessed the storage heap.	`"Mytest" 1.0 has just tried to write to the storage heap and that's just plain not allowed! Try using DmWrite.`
Stack overflow	The application pushed more information onto the stack than is allocated for the stack.	`"Mytest" 1.0 has just overflowed its stack.`
Stack almost overflowed	The stack is close to overflowing, which means that the stack pointer is within a small number of bytes (typically 100) of the end of the stack.	`"Mytest" 1.0 is getting close to overflowing the stack.`

Table 1.14 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
Memory Manager sempahore aqcuisition time	The application or system software has acquired the Memory Manager semaphore for write access, and has held it for more than 10 milliseconds.	`"Mytest" 1.0 has held the "Memory Manager semaphore" for approximately 20 milliseconds. It is recommended that applications not hold the semaphore longer than 10 milliseconds.`

Invalid heap	Heap corruption detected during a regular heap check. The Palm OS Emulator regularly checks the heap.	`During a regular checkup, the Emulator determined that the dynamic heap got corrupted.`
		
		`(corruption type) is one of the following message fragments: · The chunk was not within the heap it was supposed to be · The size of the chunk (chunk_size) was larger than the currently accepted maximum (chunk_max) · Some unused flags were set to "1" · The "hOffset" field of the chunk header did not reference a memory location within a master pointer block · The master pointer referenced by the "hOffset" field in the chunk`

Table 1.14 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
Invalid program counter	The program counter has been set to an invalid memory location, which is a location outside of a `'CODE'` resource.	`"Mytest" 1.0 has just set the Program Counter (PC) to an invalid memory location.`
Unimplemented trap.	The application or system software has attempted to invoke an unimplemented system function. An unimplemented system function is one with a trap number outside of the the numbers in the system function dispatch table, or one whose table entry matches that of the `SysUnimplemented` function.	`"Mytest" 1.0 tried to call Palm OS routine trapNum (trapName). This routine does not exist in this version of the Palm OS.`

Table 1.14 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
SysFatalAlert	The application or system software has called the `SysFatalAlert` function. The Palm OS Emulator patches the `SysFatalAlert` function and present the message in its own dialog box, to allow the user to choose how to respond to the error.	`"Mytest" 1.0 has failed, reporting "attempted divide by 0". If this is the latest version of "Mytest”, please report this to the application author.`
Unhandled exception	The application or system software has caused an exception that the Palm OS Emulator cannot handle itself.	`"Mytest" 1.0 has just performed an illegal operation. It performed a "exception". If this is the latest version of "Mytest" 1.0, please report this to the application author.`

Sending Commands to Palm OS Emulator

You can use RPC packets to send commands to the Palm OS Emulator. You can invoke any function in the Palm OS dispatch table, including the Host Control functions, which are described in Appendix B, “Host Control API.”

The RPC packets use the same format as do packets that are sent to the debugger interface, which is described in Appendix A, “Debugger Protocol Reference.”

You use the socket defined by the `RPCSocketPort` preference to make RPC calls to Palm OS Emulator. When you send a packet to the emulator, you must set the dest field of the packet header to the value defined here:

```
#define slkSocketRPC (slkSocketFirstDynamic+10)
```

NOTE: You can disable the RPC command facility by setting the value of the `RPCSocketPort` preference to `0`.

You can send four kinds of command packets to the emulator:

- ReadMem
- WriteMem
- RPC
- RPC2

The first three packet types are described in Appendix A, "Debugger Protocol Reference." The fourth packet type, RPC2, is an extension of the RPC packet format that allows support for a wider range of operations.

The RPC2 Packet Format

```
#define sysPktRPC2Cmd   0x20
#define sysPktRPC2Rsp   0xA0

struct SysPktRPCParamInfo
{
  UInt8   byRef;
  UInt8   size;
  UInt16  data[1];
};

struct SysPktRPC2Type
{
  _sysPktBodyCommon;
  UInt16  trapWord;
  UInt32  resultD0;
  UInt32  resultA0;
  UInt16  resultException;
  UInt8   DRegMask;
  UInt8   ARegMask;
  UInt32  Regs[1];
  UInt16  numParams;
  SysPktRPCParamTypeparam[1];
```

```
};
```

Almost all of the RPC2 packet format is the same as the RPC format that is described in Appendix A, "Debugger Protocol Reference." The RPC2 packet includes the following additional fields:

`resultException`
: Stores the exception ID if a function call failed due to a hardware exception. Otherwise, the value of this field is `0`.

`DRegMask`
: A bitmask indicating which D registers need to be set to make this call.

`ARegMask`
: A bitmask indicating which A registers need to be set in rder to make this call.

`Regs[1]`
: A variable length array containing the values to be stored in the registers that need to be set.

  Only the registers that are being changed need to be supplied. Most of the time, `DRegMask` and `ARegMask` are set to zero and this field is not included in the packet.

  If more than one register needs to be set, then the register values should appear in the following order: D0, D1, ..., D6, D7, A0, A1, ..., A6, A7. Again, only values for the registers specified in `DRegMask` and `ARegMask` need to be provided.

Getting Help With Palm OS Emulator

Palm OS Emulator is constantly evolving, and Palm Computing is always interested in hearing your comments and suggestions.

Palm provides a forum (emulator-forum@ls.palm.com) for questions and comments about Palm OS Emulator.

You can the latest information about Palm OS Emulator in the Palm developer zone on the Internet:

http://www.palm.com/devzone.

NOTE: The source code for Palm OS Emulator is available on the Palm OS Emulator seed page: http://www.palm.com/devzone. You can create your own emulator by modifying this code.

For more information on the protocol used in Palm OS Emulator to send requests to and receive responses from a debugging target, see Appendix A, "Debugger Protocol Reference."

2

Using Palm Debugger

Palm Debugger is a tool for debugging Palm OS® applications. Palm Debugger is available for use on both Mac OS and Windows 95/98/NT platforms.

This chapter provides an introduction to and overview of using Palm Debugger. The commands that you can use are described in Chapter 3, "Palm Debugger Command Reference."

This chapter contains the following sections:

- About Palm Debugger provides a broad overview of the program and a description of its windows.
- Connecting to The Handheld Device describes how to connect Palm Debugger with the Palm OS Emulator or with a Palm Computing Platform handheld device.
- Using the Console and Debugging Windows Together describes how to use the menus and keyboard to send commands to the handheld device from the debugging and console windows.
- Using the Debugging Window and Using the Source Window describe the command and display capabilities available in each of these windows. The debugging window section also includes a full description of Using Debugger Expressions.
- Palm Debugger Error Messages describes how to decode the error messages you receive from Palm Debugger.
- Palm Debugger Tips and Examples provides a collection of tips to make your debugging efforts easier and examples of performing common debugging tasks.

About Palm Debugger

Palm Debugger provides source and assembly level debugging of Palm OS executables, and includes the following capabilities:

- support for managing Palm OS databases
- communication with Palm handheld devices
- communication with Palm OS Emulator, the Palm emulation program
- command line interface for system administration on Palm handheld devices

NOTE: You can use Palm Debugger with a Palm Computing platform handheld device, or with the Palm OS Emulator program. Debugging is the same whether you are sending commands to the emulator or to actual hardware. Connecting with either a handheld device or the Emulator is described in Connecting Palm Debugger With a Target.

Palm Debugger provides two different interfaces that you can use to send commands from your desktop computer to the handheld device:

- The console interface is provided by the *console nub* on the handheld device. You can connect to the console nub and then send console commands to the nub from Palm Debugger's console window. The console commands are used primarily for administration of databases on the handheld device.

  The console can also be used with the Palm Simulator and the CodeWarrior Debugger, and is documented in a separate chapter. For more information about the console window and the console commands, see Chapter 4, "Using the Console Window."

- The debugging interface is provided by the *debugger nub* on the handheld device. You can attach to the debugger nub and then send debugging commands to the debugger nub from Palm Debugger's debugging window. For more information about using the debugging window and the debugging commands, see Using the Debugging Window.

The console window and the debugging window each has its own set of commands that you can use to interface with the handheld device. The debugging commands are described in Chapter 3, "Palm Debugger Command Reference.", and the console window commands are described in Chapter 4, "Using the Console Window."

NOTE: The Palm OS Emulator emulates the console and debugging nubs, which allows Palm Debugger to send the same commands to the Emulator as it does to a handheld device.

On certain platforms, Palm Debugger also provides a multi-pane source window for source-level debugging. You can use this window if you have compiled your program with certain compilers that generate an appropriate symbol file. Table 2.1 summarizes the Palm Debugger windows.

Table 2.1 Palm Debugger windows

Window name	Usage
Console	Command language shell for performing administrative tasks, including database management, on the handheld device.
CPU Registers	Assembly language debugging output only window.
Debugging	Assembly language debugging command window.
Source	Source level debugging window. **NOTE:** Source level debugging is not currently available in the Macintosh version of palm Debugger.

Connecting Palm Debugger With a Target

You can use Palm Debugger to debug programs running on a Palm Computing Platform handheld device or to debug programs running on a hardware emulator such as the Palm OS Emulator. This section describes how to connect the debugger to each of these targets.

Connecting to The Palm OS® Emulator

You can interact with the Palm OS Emulator from Palm Debugger just as you do with actual hardware. With the emulator, you don't need to activate the console or debugger stubs. All you need to do is follow these steps:

1. In the Palm Debugger Communications menu, select **Emulator**. This establishes the emulator program as the "device" with which Palm Debugger is communicating.
2. In the debugger window, type the `att` command.

Connecting to The Handheld Device

You can interact with the handheld device from Palm Debugger by issuing commands from the console window or from the debugging window.

You must activate the console nub on the handheld device before sending commands from the console window. For more information on activating console input, see Chapter 4, "Using the Console Window."

WARNING! When you activate either the console nub or the debugger nub on the handheld device, the device's serial port is opened. This causes a rapid and significant power drain. The only way to close the port and stop the power drain is to perform a soft reset.

Activating Debugging Input

If you are debugging with the Palm OS Emulator, you can activate debugging input by sending the `att` command from the debugging window to the emulator.

To send debugging commands to a hardware device, you must connect your desktop computer to the handheld device, halt the device in its debugger nub, and then type commands into the debugging window of Palm Debugger.

IMPORTANT: When the handheld device is halted in its debugger nub, a tiny square flashes in the upper left corner of the screen, and the device does not respond to pen taps or key presses.

You can use the following methods to halt the handheld in its debugger nub:

1. Use the Graffiti Shortcut-1 to enter debugger mode on the handheld device, as described in Using Shortcut Numbers to Activate the Windows.
2. If you have already activated the console nub, you can use the **Break** command in the Source menu to activate the debugger nub. The **Break** command sends a key command to the handheld device that is identical to using the Graffiti Shortcut-1 sequence.
3. Compile a `DbgBreak()` call into your application, and run the application until you encounter that call.

   This method only works if you have already entered debugger mode once, or if you have set the low memory global variable `GDbgWasEntered` to a non-zero value, which tricks the handheld into thinking that the debugger was previously entered. For example, you can use the following code in your application to ensure that your break works:

   ```
   GDbgWasEntered = true;
   DbgBreak();
   ```

4. You can hold the down button and press the reset button in the back of the device.

   This halts the device in the *SmallROM* debugger, which is the bootstrap code that can initialize the hardware and start the debugger nub. Enter the `g` command, and the system jumps into the *BigROM*, which contains the same code as the SmallROM and all of the system code.

   If you press the down button on the handheld device while executing the `g` command, you land in the BigROM's debugger. This allows you to set A-trap breaks or single step through the device boot sequence.

Verifying Your Connection

If Palm Debugger is running and connected when the handheld device halts into its debugger nub, the debugging window displays a message similar to the following:

```
EXCEPTION ID = $A0
'SysHandleEvent'
  +$0512  10C0EEFE  *MOVEQ.L   #$01,D0 | 7001
```

Alternatively, if Palm Debugger is not connected or running when the device halts, you can use the `att` command to attach Palm Debugger to the device.

WARNING! The debugger nub activates at 57,600 baud, and your port configuration must match this is you are connecting over a serial port. You can set the connection parameters correctly with Palm Debugger Connection menu.

After you activate the debugger nub on the handheld device, the nub prevents other applications, including HotSync® from using the serial port. You have to soft-reset the handheld device before the port can be used.

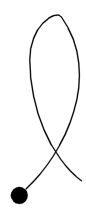
Using Shortcut Numbers to Activate the Windows

The Palm OS® responds to a number of "hidden" shortcuts for debugging your programs, including shortcuts for activating the console and debugger nubs on the handheld device. You generate each of these shortcuts by drawing characters on your Palm Computing platform device, or by drawing them in the Palm OS Emulator program, if you are using Palm OS Emulator to debug your program.

NOTE: If you open the Find dialog on the handheld device before entering a shortcut number, you get visual feedback as you draw the strokes.

To enter a shortcut number, follow these steps:

1. On your Palm Computing platform device, or in the emulator program, draw the shortcut symbol. This is a lowercase, cursive "L" character, drawn as follows:



2. Next, tap the stylus twice, to generate a dot (a period).
3. Next, draw a number character in the number entry portion of the device's text entry area. Table 2.2 shows the different shortcut numbers that you can use.

   For example, to activate the console nub on the handheld device, enter the follow sequence:

 .2

Table 2.2 Shortcut numbers for debugging

Shortcut	Description	Notes
 .1	The device enters debugger mode, and waits for a low-level debugger to connect. A flashing square appears in the top left corner of the device.	This mode opens a serial port, which drains power over time. You must perform a soft reset or use the debugger’s `reset` command to exit this mode.
 .2	The device enters console mode, and waits for communication, typically from a high-level debugger.	This mode opens a serial port, which drains power over time. You must perform a soft reset to exit this mode.
 .3	The device’s automatic power-off feature is disabled.	You can still use the device’s power button to power it on and off. Note that your batteries can drain quickly with automatic power-off disabled. You must perform a soft reset to exit this mode.

NOTE: These debugging shortcuts leave the device in a mode that requires a soft reset. To perform a soft reset, press the reset button on the back of the handheld with a blunt instrument, such as a paper clip.

Using the Console and Debugging Windows Together

When Palm Debugger is attached to a handheld device or emulator program, you cannot talk to the console nub on the device. However, a subset of the console commands — those that do not change the contents of memory— are available from the debugging window. These include the following commands:

- `dir`
- `hl`
- `hd`
- `hchk`
- `mdebug`
- `reset`

You can enter these commands in either the debugging window or the console window when the debugger nub is active. When you enter a console command while the debugging window is attached, the command is sent to the debugger nub rather than the console nub.

You can use the console commands while debugging for purposes such as displaying a heap dump in the console window while stepping through code in the debugging window.

Entering Palm Debugger Commands

Most of your work with Palm Debugger is done with the keyboard. You enter console commands into the console window, and debugging commands into the debugging window. Both of these windows supports standard scrolling and clipboard operations.

Table 2.3 summarizes the keyboard commands that you can use to enter commands in Palm Debugger's console or debugging windows.

Table 2.3 Entering Palm Debugger Commands From the Keyboard

Command description	Windows key(s)	Macintosh key(s)
Execute selected text as command(s). You can select multiple lines to sequentially execute multiple commands. Execute the current line (no text selected).	**Enter**	**Enter** on numeric keypad, or **Cmd-Return**
Display help for a command	**Help <cmdName>**	**Help <cmdName>**
Enter a new line without executing the text	**Shift-Enter**	**Return**
Copy selected text from window to clipboard	**Ctrl-C**	**Cmd-C**
Paste clipboard contents to window	**Ctrl-V**	**Cmd-V**
Cut selected text from window to clipboard	**Ctrl-X**	**Cmd-X**
Delete previous command's output from the window	**Ctrl-Z**	not available
Delete all text to the end	**Shift-Backspace**	**Cmd-delete**

Palm Debugger Menus

Palm Debugger includes five menus, as summarized in Table 2.4. The most commonly used menu commands are on the Connection and Source menus; these commands are described in other sections in this chapter.

Table 2.4 Palm Debugger menu commands

Menu	Commands	Descriptions
File	**Open** **Save** **Save As**	Commands for saving and printing the contents of a window.
	Page Setup... **Print**	
	Exit	
Edit	**Undo** **Redo**	Standard editing commands
	Cut **Copy** **Paste** **Select All**	
	Find **Find Next**	
	Font	
Connection	(select baud rate)	For setting up how to communicate with the handheld device or Palm OS Emulator.
	Handshake	
	(select connection port)	

Table 2.4 Palm Debugger menu commands *(continued)*

Menu	Commands	Descriptions
Source	**Break** **Step Into** **Step Over** **Go** **Go Till** **Toggle Breakpoint** **Disassemble at Cursor** **Show Current Location** **Install Database and Load Symbols** **Load Symbols** **Load Symbols for Current Program Counter** **Remove All Symbols**	Source code debugging commands, for use in conjunction with the source window. **NOTE:** Source level debugging is not currently available in the Macintosh version of Palm Debugger.
Window	**Cascade** **Tile** **Arrange Icons** **Close All** **Keyboard Simulator...** (select numbered window)	Standard window access commands. **NOTE:** Only available on Windows systems.

Palm Debugger Command Syntax

Palm Debugger's help facility uses simple syntax to specify the format of the commands that you can type in the console and debugging windows. This same syntax is used in Chapter 3, "Palm Debugger Command Reference." This section summarizes that syntax.

The basic format of a command is specified as follows:

```
commandName <parameter>* [options]
```

`commandName` The name of the command.

`parameter`	Parameter(s) for the command. Each parameter name is enclosed in angle brackets (< and >). Sometimes a parameter can be one value or another. In this case the parameter names are bracketed by parentheses and separated by the **|** character.
`options`	Optional flags that you can specify with the command. Note that options are specified with the dash (`-`) character in the console window and with the backslash (`\`) character in the debugging window.

NOTE: Any portion of a command that is shown enclosed in square brackets ("[" and "]") is optional.

The following is an example of a command definition

```
dir (<cardNum>|<srchOptions>) [displayOptions]
```

The `dir` command takes either a card number of a search specification, followed by display options.

Here are two examples of the `dir` command sent from the console window:

```
dir 0 -a
dir -t rsrc
```

And here are the same two commands sent from the debugging window:

```
dir 0 \a
dir \t rsrc
```

Specifying Command Options

All command options and some command parameters are specified as flags that begin with a dash (in the console window) or backslash (in the debugging window). For example:

```
-c
-enable
```

```
\enable
```

Some flags are followed by a keyword or value. You must leave white space between the flag and the value. For example:

```
-f D:\temp\myLogFile
\t Rsrc
```

Specifying Numeric and Address Values

Many of the debugging commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (#) or binary (%). Table 2.5 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 2.5 Specifying numeric values in Palm Debugger

Hex value	Decimal value	Binary value
`64 or $64`	`#100`	`%01100100`
`F5 or $F5`	`#245`	`%11110101`
`100 or $100`	`#256`	`%100000000`

WARNING! Some register names, like A0 and D4, look like hexadecimal values. You must preface these values with the $ sign, or you will get the value of the register. For example, A4 + 3 computes to the value of the A4 register added with three, but $A4 + 3 computes to $A7.

For more information, see Specifying Constants.

Using the Debugging Window

You use the debugging window to enter debugging commands, which are used to perform assembly language debugging of executables on the handheld device. Commands that you type into the debugging window are sent to the debugger nub on the handheld device, and the results sent back from the device are displayed in the debugging window.

The debugging window provides numerous capabilities, including the following:

- A rich expression language for specifying command arguments, as described in Using Debugger Expressions.
- Ability to debug applications, system code, extensions, shared libraries, background threads, and interrupt handlers.
- Custom aliases for commands or groups of commands, as described in Defining Aliases.
- Script files for saving and reusing complex sequences of commands, as described in Using Script Files.
- Templates for defining data structure layouts in memory, which allow you to view a structure with the memory display commands. Templates are described in Defining Structure Templates.
- Your aliases and templates can be saved in files that are automatically loaded for you when Palm Debugger starts execution, as described in Automatic Loading of Definitions.

This section also provides examples of using some of the more common debugging commands:

- See Displaying Registers and Memory for examples of using the debugging commands to display the current register values.
- See Using the Flow Control Commands for examples of using commands to set breakpoints.
- See Using the Heap and Database Commands for examples of using commands to examine the heap and databases.

The remainder of this section describes how to use these capabilities. Table 2.6 shows the most debugging window command categories.:

Table 2.6 Debugging window command categories

Category	Description	Commands
Console	Commands shared with the console window for viewing card, database, and heap information.	`cardinfo, dir, hchck, hd, hl, ht, info, opened, storeinfo`
Flow Control	Commands for working with breakpoints, A-traps, and program execution control.	`atb, atc, atd, br, brc, cl, brd, dx, g, gt, s, t, reset`
Memory	Commands for viewing the registers, and for displaying and setting memory, the stack, and system function information.	`atr, db, dl, dm, dw, fb, fill, fl, ft, fw, il, reg, sb, sc, sc6, sc7, sl, sw, wh`
Miscellaneous	Commands for displaying debugging help and current debugging environment information.	`att, help, penv`
Template	Commands for defining and reviewing structure templates.	`>, sizeof, typedef, typeend`
Utility	Commands for working with aliases, symbol files, and variables.	`alias, aliases, bootstrap, keywords, load, run, save, sym, templates, var, variables`

All of the debugging commands are described in detail in Chapter 3, "Palm Debugger Command Reference."

Before you can use the debugging commands, you must attach Palm Debugger to the debugger nub on the handheld device, as described in Activating Debugging Input.

Using Debugger Expressions

Palm Debugger provides a rich expression language that you can use when specifying arguments to the debugging commands. This section describes the expression language.

NOTE: Debugger expressions cannot contain white space. White space delimits command parameters; thus, any white space ends an expression.

Specifying Constants

The expression language allows you to specify numbers as character constants.

Character Constants

A character is a string enclosed in single quotes. The string can include escape sequences similar to those used in the C language. For example:

```
'xyz1'
'a\'Y\''
'\123'
```

Character constants are interpreted as unsigned integer values. The size of the resulting value depends on the number of characters in the string, as follows:

Number of characters	Result type
1 character	`UInt8`
2 characters	`UInt16`
more than 2 characters	`UInt32`

Binary Numbers

To specify a binary number, use the `%` character followed by any number of binary digits. For example:

```
%00111000
%1010
```

The size of the resulting value is determined as follows:

Number of digits	Result type
1 to 8	`UInt8`
8 to 16	`UInt16`
more than 16	`UInt32`

Decimal Numbers

To specify a decimal number, use the # character followed by any number of decimal digits. For example:

```
#256
#32756
```

Hexadecimal Numbers

Palm Debugger interprets hexadecimal digit strings that are not preceded by a special character as hexadecimal numbers. You can optionally use the `$` character to indicate that a value is hexadecimal. For example:

```
c123
C123
F0
$A0
```

The size of the resulting value is determined as follows:

Number of digits	Result type
1 to 2	`UInt8`
3 to 4	`UInt16`
more than 4	`UInt32`

WARNING! If you want to specify a hexadecimal value that can also be interpreted as a register name, you must preface the value with the $ symbol. For example, using `A0` in an expression will generate the current value of the `A0` register, while using $A0 will generate the hexadecimal equivalent of the decimal value `160`.

Using Operators

Palm Debugger expression language includes the typical set of binary and unary operators, as summarized in Table 2.7.

Table 2.7 Palm Debugger expression language operators

Type	Operator	Description	Example
Cast	`.a`	Casts the value to an address.	`0ff0.a`
	.b	Casts the value to a byte.	45.b
	`.l`	Casts the value to a double word.	`45.l`
	`.w`	Casts the value to a word.	`45.w`
	`.s`	Extends the sign of its operand without changing the operand's size.	`45.s`
Unary	`~`	Performs a bitwise NOT of the operand.	`~1`
	`-`	Changes the sign of the operand.	`2*-1`
Dereference	`@`	Dereferences an address or integer value. See Table 2.8 for more examples.	@A7
Arithmetic	`*`	Multiplies the two operands together.	`A1*2`
	`/`	Divides the first operand by the second operand.	`21/3`
	`+`	Adds the two operands together.	`A2+2`
	`-`	Subtracts the second operand from the first operand.	A2-2

Table 2.7 Palm Debugger expression language operators

Type	Operator	Description	Example
Assignment	=	Assigns the second operand value to the register specified as the first operand.	d0=45
Bitwise	&	Performs a bitwise AND operation.	`A7&FFF`
	^	Performs a bitwise XOR operation.	`A2^F0F0`
	|	Performs a bitwise OR operation.	`A2|%1011`

The Dereference Operator

The @ dereference operator is similar to the * dereference operator used in the C programming language. This operators dereferences an address value, as shown in Table 2.8.

Table 2.8 Dereference operator examples

Expression	Description	Example
`@`	Retrieves 4 bytes as an unsigned integer value	`@A7`
`@.a`	Retrieves 4 bytes as an address	`@.a(A1)`
`@.b`	Retrieves 1 byte as an unsigned integer value	`@.b(PC)`
`@.w`	Retrieves 2 bytes as an unsigned integer value	`@.w(PC)`
`@.l`	Retrieves 4 bytes as an unsigned integer value	`@.l(A2)`

Register Variables

The expression language provides named variables for each register. The names of these variables are replaced by their respective register values in any expression. Table 2.9 shows the register name variables.

Table 2.9 The built-in register variables

Variable name	Description
`a0`	address register 0
`a1`	address register 1
`a2`	address register 2
`a3`	address register 3
`a4`	address register 4
`a5`	address register 5
`a6`	address register 6
`a7`	address register 7
`d0`	data register 0
`d1`	data register 1
`d2`	data register 2
`d3`	data register 3
`d4`	data register 4
`d5`	data register 5
`d6`	data register 6
`d7`	data register 7
`pc`	the program counter
sr	the status register
sp	the stack pointer (this is an alias for `a7`)

NOTE: The expression parser interprets any string that can represent a register name as a register name. If you want the string interpreted as a hexadecimal value instead, precede it with either a `0` or the `$` character.

For example, the following expression:

```
a0+d0
```

Adds the values stored in the a0 and d0 registers together.

If you want to add the value `0xd0` to the value stored in register a0, use one of the following expressions:

```
a0+0d0
a0+$d0
```

Special Shortcut Characters

Palm Debugger's expression language includes the two special value characters show in Table 2.10. These characters are converted into values in any expression.

Table 2.10 Special value expression characters

Character	Converts into	Examples
`.`	The most recently entered address.	`dm .` `dm .+10`
`:`	The starting address of the current routine.	`il :` `il :+24`

Performing Basic Debugging Tasks

This section describes how to use Palm Debugger to perform three of the most common debugging tasks:

- displaying memory values
- setting breakpoints and using the flow control commands
- examining the heap

The final section of this chapter, Palm Debugger Tips and Examples, provides examples of how to perform other debugging tasks.

Assigning Values to Registers

You can use the assignment operator (=) to assign a value to a register. However, if you include white space around the operator, the assignment does not work. For example, the following statement correctly assigns a value to the program counter:

```
pc=010c8954
```

However, this statement does not assign the correct value to the program counter:

```
pc = 010c8954c
```

Displaying Registers and Memory

One of the primary operations you perform with a debugger is to examine and change values in memory. Palm Debugger provides a number of commands for displaying registers, memory locations, the program counter, and the stack. Table 2.11 summarizes the commands you commonly use to examine memory and related values.

Table 2.11 Frequently used memory commands

Command	Description
`db`	Displays the byte value at a specified address.
`dl`	Displays the 32-bit long value at a specified address.
`dm`	Displays memory for a specified number of bytes or templates.
`dw`	Displays the 16-bit word value at a specified address.
`il`	Disassembles code in a specified line range or for a specified function name.
`reg`	Displays all registers.
`sb`	Sets the value of the byte at the specified address.

Table 2.11 Frequently used memory commands *(continued)*

Command	Description
`sc`	Lists the A6 stack frame chain, starting at the specified address.
`sc7`	Lists the A7 stack frame chain, starting at the specified address.
`sl`	Sets the value of the 32-bit long value at the specified address.
`sw`	Sets the value of the word at the specified address.

Palm Debugger also allows you to define structure templates and use those for displaying memory values. For example, you can define a structure that matches the layout of a complex data structure, and then display that structure with a single `dm` command. For more information about structure templates, see Defining Structure Templates.

Listing 2.1 shows an example of displaying memory with the `dm` command and disassembling memory with the `il` command. It also provides several examples of using expressions with these commands. In this example, **`boldface`** is used to denote commands that you type, and <= starts a comment.

Listing 2.1 Displaying and disassembling memory

```
dm 0                  <=Display memory at address 0
00000000: FF FF FF FF 1A 34 3E 40 10 C0 92 D4 10
C0 92 F2 ".....4>@........"

dm 100                <=Display memory at address
0x100
00000100: 01 01 00 00 02 B0 00 01 78 30 00 00 00
01 47 EE "........x0....G."

dm #100               <=Display memory at address 100
decimal
```

```
00000064: 10 C6 BE 32 10 C6 BE 60 10 C6 BE 8E 10
C6 BE BC "...2...`........"

dm 100+20          <=Specify an address with an
expression
00000120: 6F BC 00 00 07 22 00 00 00 06 00 01 7D
72 00 FD "o....".......}r.."

dm .+10            <=Use the'.' character for the
most recent addr
00000130: 00 00 00 00 00 00 00 B6 3E C0 69 45 A4
0C 03 4A "........>.iE...J"

dm pc              <=Use the current program
counter value
10C0EEFE: 70 01 60 00 01 7E 4E 4F A0 BE 70 01 60
00 01 74 "p.`..~NO..p.`..t"

dm pc+20           <=An expression using the
program counter
10C0EF1E: FF F4 4E 4F A0 AC 38 00 4A 44 50 4F 66
2A 48 6E "..NO..8.JDPOf*Hn"

il pc              <=Disassemble code at current
program counter
'SysHandleEvent 10C0E9EC'
+$0512 10C0EEFE *MOVEQ.L #$01,D0 | 7001
+$0514 10C0EF00 BRA.W SysHandleEvent+$0694 ;
10C0F080         | 6000 017E
+$0518 10C0EF04 _SysLaunchConsole ; $10C0E30C |
4E4F A0BE
+$051C 10C0EF08 MOVEQ.L #$01,D0 | 7001
+$051E 10C0EF0A BRA.W SysHandleEvent+$0694 ;
10C0F080         | 6000 0174
+$0522 10C0EF0E MOVEQ.L #$00,D0 | 7000
+$0524 10C0EF10 BRA.W SysHandleEvent+$0694 ;
10C0F080         | 6000 016E
+$0528 10C0EF14 CLR.L -$0010(A6) | 42AE FFF0
+$052C 10C0EF18 PEA -$0006(A6) | 486E FFFA
+$0530 10C0EF1C PEA -$000C(A6) | 486E FFF4
```

```
il pc-10           <=Display code at program
counter - 0x10
'SysHandleEvent 10C0E9EC'
+$0502 10C0EEEE ORI.B #$01,(A5)+ ; '.' | 001D 7001
+$0506 10C0EEF2 BRA.W SysHandleEvent+$0694 ;
10C0F080          | 6000 018C
+$050A 10C0EEF6 MOVE.B #$01,$00000101 ; '.' | 11FC
0001 0101
+$0510 10C0EEFC _DbgBreak | 4E48
+$0512 10C0EEFE *MOVEQ.L #$01,D0 | 7001
+$0514 10C0EF00 BRA.W SysHandleEvent+$0694 ;
10C0F080          | 6000 017E
+$0518 10C0EF04 _SysLaunchConsole ; $10C0E30C |
4E4F A0BE
+$051C 10C0EF08 MOVEQ.L #$01,D0 | 7001
+$051E 10C0EF0A BRA.W SysHandleEvent+$0694 ;
10C0F080          | 6000 0174
+$0522 10C0EF0E MOVEQ.L #$00,D0 | 7000
```

All of the commands mentioned in this section are described in detail in Chapter 3, "Palm Debugger Command Reference."

Using the Flow Control Commands

Palm Debugger provides a number of commands for setting breakpoints and continuing the flow of execution. Table 2.12 summarizes the commands you commonly use for these purposes.

Table 2.12 Commonly used flow control commands

Command	Description
`atb`	Adds an A-trap break.
`atc`	Clears an A-trap break.
`atd`	Displays all A-trap breaks.
`br`	Sets a breakpoint.
brc	Clears a breakpoint. This is the same as the `cl` command.

Table 2.12 Commonly used flow control commands

Command	Description
`brd`	Display all breakpoints.
`cl`	Clears a breakpoint. This is the same as the `brc` command.
`g`	Continues execution until the next breakpoint is encountered.
`gt`	Sets a temporary breakpoint at the specified address, and resumes execution from the current program counter.
`s`	Single steps one source line, stepping into functions.
ss	Step-spy: step until the value of the specified address changes.
`t`	Single steps one source line, stepping over functions.

Listing 2.2 shows an example of setting breakpoints, disassembling, and using other flow control commands to debug an application. In this example, **`boldface`** is used to denote commands that you type, and <= starts a comment.

Listing 2.2 Using the debugging flow control commands

```
sc              <= Display stack crawl, listed from
oldest to newest. In this
                <= example, the current fcn was
called from EventLoop+0016
Calling chain using A6 Links:
 A6 Frame   Caller
 00000000  10C68982  cjtkend+0000
 00015086  10C6CA26  __Startup__+0060
 00015066  10C6CCCE  PilotMain+0250
 00014FC2  10C0F808  SysAppLaunch+0458
 00014F6E  10C10258  PrvCallWithNewStack+0016
```

```
 00013418  10CD88B2  __Startup__+0060
 000133F8  10CDB504  PilotMain+0036
 000133DE  10CDB47C  EventLoop+0016

s                <= Single-Step one instruction
'SysHandleEvent' Will Branch
+$0514 10C0EF00 *BRA.W SysHandleEvent+$0694 ;
10C0F080          | 6000 017E
                 <= Single step again by pressing
theEnter key
+$0694 10C0F080 *MOVEM.L (A7)+,D3-D5/A2-A4 | 4CDF
1C38
                 <= Press enter again
+$0698 10C0F084 *UNLK A6 | 4E5E
                 <= ... and again
+$069A 10C0F086 *RTS | 4E75 8E53 7973 4861
                 <= ... and again
+$0018 10CDB47E *TST.B D0 | 4A00

il               <= Disassemble at current program
counter
'EventLoop 10CDB466'
+$0018 10CDB47E *TST.B D0 | 4A00
+$001A 10CDB480 LEA $000C(A7),A7 | 4FEF 000C
+$001E 10CDB484 BNE.S EventLoop+$0050 ; 10CDB4B6 |
6630
...              <= Remainder of disassembly
removed here

gt 10cdb484      <= Go-Till address 0x10CDB484
+$001E 10CDB484 *BNE.S EventLoop+$0050 ; 10CDB4B6
| 6630

br :+50          <= Set a breakpoint at current
routine+0x50
Breakpoint set at 10CDB4B6 (EventLoop+0050)

g                <= Go until a break occurs
+$0050 10CDB4B6 *CMPI.W #$0016,-$0018(A6) ; '..' |
0C6E 0016 FFE8
```

```
brd               <= Display all currently set
breakpoints
10CDB4B6 (EventLoop+0050)

cl                <= Clear all breakpoints
All breakpoints cleared

atb "EvtGetEvent" <= Break whenever the
EvtGetEvent system trap is called
A-trap set on 011d (EvtGetEvent)

g                 <= Go until a break occurs
Remote stopped due to: A-TRAP BREAK EXCEPTION
'EvtGetEvent'
+$0000 10C3B1E2 *LINK A6,$0000 | 4E56 0000

atc               <= Clear all A-Traps
All A-Traps cleared

ss a2             <= Step-Spy until the UInt32 at
address 0x15404 changes
                  <= (the current value of register
A2 is 0x15404)
Step Spying on address: 00015404
'EvtGetSysEvent'
  +$00E8  10C1E980  *CLR.B    $0008(A4)
| 422C 0008
```

WARNING! Some commands, like the `atb` command, require that the operand be quoted. Forgetting to quote the trap name in the `atb` command is a common mistake with Palm Debugger.

All of the commands mentioned in this section are described in detail in Chapter 3, "Palm Debugger Command Reference."

Using the Heap and Database Commands

You can use the heap and database commands to display information about the databases and heaps on the handheld device. These commands, which are summarized in Table 2.13, mirror commands available from the console window.

Table 2.13 Commonly used heap and database commands

Command	Description
`dir`	Lists the databases.
`hchk`	Checks a heap.
`hd`	Displays a dump of a memory heap.
`hl`	Lists all of the memory heaps on the specified memory card.
`ht`	Performs a heap summary.

The heap commands take heap ID values as parameters. The following table shows the values you can use for heap IDs.

Heap ID	Description
0	The dynamic heap.
1	The storage heap.

All of the commands mentioned in this section are described in detail in Chapter 3, "Palm Debugger Command Reference."

To learn more about the console window and all of the console commands, see Using the Console Window.

Advanced Debugging Features

This section presents several advanced features of the debugging window of Palm Debugger, including the following:

- defining structure template for displaying memory
- defining aliases for commands
- using script files to run sequences of commands

- automated loading of structure and alias definitions at program start-up time

Defining Structure Templates

You can define structure templates to use with Palm Debugger's memory display commands. Each template matches a data type or structure type that you use in your application, which allows you to display a structure in the debugging window with one command.

You define templates in a manner similar to the way you define structure types in a high-level programming language: start a template definition with the `typedef` command, follow with some number of field definition (>) commands, and finish with a `typeend` command. And once you have defined a structure template, you can use fields of that type in other template definitions.

Table 2.14 summarizes the commands you use to define and display templates. For more information about these commands, see Chapter 3, "Palm Debugger Command Reference."

Table 2.14 Structure template commands

`>`	Defines a structure field.
`sizeof`	Displays the size, in bytes, of a template.
`templates`	Lists the names of the debugger templates.
`typedef`	Begins a structure definition block.
`typeend`	Ends a structure definition block.

Note that the structure and field names must be quoted in your structure template definition commands. Listing 2.3 shows the debugging commands used to define a template named `PointType`, and then defines a second template named `RectangleType` that uses two `PointType` fields.

Listing 2.3 Defining and using two structure templates

```
typedef struct "PointType"
> Int16 "X"
```

```
> Int16 "Y"
typeend


typedef struct "RectangleType"
> PointType "topLeft"
> PointType "extent"
typeend

sizeof PointType

Size = 4 byte(s)

sizeof RectangleType
Size = 8 byte(s)

dm 0 RectangleType
00000000  struct    RectangleType
            {
00000000    PointType topLeft
              {
00000000      Int16 x         = $-1
00000002      Int16 y         = $-1
              }
00000004    PointType extent
              {
00000004      Int16 x         = $1A34
00000006      Int16 y         = $3E40
              }
            }
```

Defining Aliases

For convenience, you can create aliases. Each alias stands for a specific command sequence. For example:

```
alias "checkheap" "hchk 0 -c"
alias "ls" "dir 0"
```

After defining these aliases, you can type `ls` to display a directory listing for card 0 (built-in RAM), and you can type `checkheap` to check heap `0` with examination of each chunk.

Using Script Files

You use the run command to run a script file. A script file is any text file that contains debugging commands. For example, the following command reads and executes the debugging commands found in the text file named `MyCommands`:

```
run "MyCommands"
```

Automatic Loading of Definitions

When Palm Debugger is launched, it automatically runs the script file named `UserStartupPalmDebugger`. You can store your aliases, script files, and data structure templates in this file to have them available whenever you use Palm Debugger.

Using the Source Window

This section describes the source window, which you can use to perform limited debugging with the source code for your application.

IMPORTANT: Palm Debugger's source level debugging is only available on Windows systems, and is only available for code that has been built using the GNU gcc compiler for Palm OS.

The source window works in conjunction with the debugging and CPU registers windows. For example, if you single step in the debugging window, the source window tracks along and displays any breakpoints that are currently set.

The source window is split into two panes:

- The upper pane displays the values of local variables for the current function.
- The lower pane displays the source code. This pane is automatically updated whenever you move through your

code with flow control commands. You can also scroll this pane to view the code or to set a breakpoint.

The left margin of the lower pane displays indicators for breakpoints and the current program counter:

- a solid red circle is displayed next to a line that contains a breakpoint
- a green arrow is displayed next to the line containing the current program location

The two panes in the source window are separated by a thick horizontal line. This line is colored red when the connected handheld device is halted in the debugger nub, and is green when the handheld device is running code.

Debugging With the Source Window

To debug with the source code for an executable, you need to associate a symbol file on your desktop computer with the executable that is running on the handheld device. You can load any number of symbol files into Palm Debugger at once; whenever the device stops in the debugger nub, Palm Debugger automatically determines which symbol file to display in the source window.

You can use the following steps to load an application and its symbol file, and then use the source debugging commands:

1. Activate the console nub, as described in Activating Console Input.
2. Select the **Install Database and Load Symbols** menu command from the Source menu.
3. Select the `.PRC` file to load onto the device.
4. Palm Debugger imports the `.PRC` file into the handheld device and looks in the same directory for the associated symbol file.

Palm Debugger now associates the symbol file with the application that has been imported into the handheld device. Whenever the debugger nub breaks in the code for that application, the source window displays the associated source file and line number.

You can also break into the debugger manually and set a breakpoint on specific source code lines with the **Toggle Breakpoint** command in the Source menu or on the source window's context menu.

Using Symbol Files

This section provides information about symbol files. You need to have a symbol file for your executable to use Palm Debugger's source code debugging facility.

Each symbol file represents a single code resource and is created by the linker. Most Palm OS applications contain a single code resource of type `'code'` and a resource ID of `1`. Some applications have more than one code resource, and thus more than one symbol file.

A symbol file contains the following items:

- the names of each of the source files that were linked together to create the code resource
- the offset from the start of the code resource to the object code for each source file
- the offset from the start of the code resource for each line in the source file
- descriptions of the data structures used
- descriptions of the name, type, and location of each local variable used in the source code's functions
- descriptions of the name, type, and location of each global variable

To make use of a symbol file, Palm Debugger needs the address of the code resource on the handheld device that corresponds to the symbol file. The **Load Symbols** command on the Source menu associates a symbol file on the desktop computer with a code resource on the handheld device.

Using the Source Menu

Palm Debugger's Source menu contains commands that you can use for source level debugging. Table 2.15 summarizes these commands. Note that several of these commands are also available from the Source context menu, as described in the next section.

Table 2.15 Source menu commands

Command	Description
Break	Halts the handheld device in the debugger nub by sending the same key event as does the Graffiti Shortcut-1 shortcut. The device must be running the console nub to activate this command.
Step Into	Single steps one source line, and stops if it steps into a subroutine.
Step Over	Single steps one source line. If it steps into a subroutine, doesn't stop until the subroutine returns.
Go	Continues execution until a breakpoint is encountered.
Go Till	Sets a temporary breakpoint at the currently selected line in the source window and then continues execution.
Toggle Breakpoint	Toggles a breakpoint on or off at the currently selected line in the source window.
Disassemble at Cursor	Disassembles code at the currently selected line in the source window. The disassembled output is displayed in the debugging window.
Show Current Location	Scrolls the source window to show the current line in the source file.
Install Database and Load Symbols	Imports a `.PRC` file into the handheld device and looks in the same directory for the associated symbol file.
Load Symbols	Opens a symbol file for use by Palm Debugger.

Table 2.15 Source menu commands *(continued)*

Command	Description
Load Symbols for Current Program Counter	***NEED HELP HERE***
Remove All Symbols	Unloads any loaded symbols.

Using the Source Window Context Menu

You can activate the source context menu by right clicking your mouse in the source window. The context menu features many of the commands are available in the Source menu, including:

- **Break**
- **Go Till**
- **Toggle Breakpoint**
- **Disassemble at Cursor**
- **Show Current Location**

The context menu also lists the source files for each symbol file that is loaded. You can use this list to select which source file you want to view.

Source Window Debugging Limitations

Source level debugging is limited in the current version of PalmDebugger. Although you can perform some of your debugging with the source window, you need to keep the following limitations in mind to remember when you need to switch back to assembly language debugging:

- You cannot display a stack crawl in the source window. You need to switch to the debugging window and use the `sc` command.
- Local variables that are structures or pointers to structures display as hexadecimal addresses in the local variables pane of the source window. To view the contents of these structures, you need to use the `dm` command in the debugging window.

- You cannot view global variables in the source window.
- Local variables are only displayed in hexadecimal format.
- You cannot change the values of local variables from the source window. To change these values, you must use the `sb`, `sw`, or `sl` commands in the debugging window.

Palm Debugger Error Messages

Most of the error messages displayed by Palm Debugger are hexadecimal codes that can be difficult to understand. To determine the meaning of the message, you need to look up the code in the Palm OS header files.

Each error code is a 16-bit value, in which the upper byte represents the code manager that generated the error, and the lower byte represents the specific error code. For example, suppose that you receive the following error message from Palm Debugger:

```
### Error $00000219
```

The code manager code is `0x02`, which is the Data Manager, and the error code is `0x19`, which is `dmErrAlreadyExists`.

The manager codes are located in the `SystemMgr.h` header file. The value `0x02` is defined as `dmErrorClass`.

The specific error codes for each manager are found in the header file for that manager. For example, the value `0x19` is defined in `DataMgr.h` as `dmErrAlreadyExists`.

Palm Debugger Tips and Examples

This section provides a collection of tips and examples for working with Palm Debugger, including the following sections:

- Performing Calculations
- Saving time with Shortcut Characters and Repeating Commands
- Finding a Specific Function
- Finding Memory Corruption Problems
- Displaying Local Variables and Function Parameters

- Changing the Baud Rate Used by Palm Debugger
- Debugging Applications That Use the Serial Port
- Importing System Extensions and Libraries
- Determining the Current Location Within an Application

NOTE: Several of the examples in this section show user input mixed with the output displayed by Palm Debugger. In these cases, the user input—the commands you type—is shown in **`boldface`**.

Performing Calculations

You can type numeric expressions into the debugging window to use it as a simple hexadecimal calculator. Here are several examples of typing a numeric expression and the results displayed in the debugging window.

Typed expression	Displayed result
#20*4+3	`$00000053 #83 #83 '...S'`
20*4+3	`$83 #131 #-125 '.'`
123+ff	`$0222 #546 #546 '."'`

Shortcut Characters

Use the two shortcut characters to simplify your typing efforts: type the period (`.`) character to specify the address value used for the most recent command, or use the semicolon (`:`) character to specify the starting address of the current routine.

Repeating Commands

You can repeat several of the debugging commands by pressing the Enter key repeatedly. For example, you can type the `dm` command to display sixteen bytes of memory, and then press the Enter key to display the next sixteen bytes of memory. The `s` and `t` commands also provide this capability.

Finding a Specific Function

A typical debugging problem is that you want to single step through some problem code, but need to first find the code. This section presents four different methods that you can use to find code:

- Rebuild the application with a call to `DbgBreak` in the problem routine.
- Use debugging commands to set an A-trap break on a system call that the problem routine makes.
- Use the `ft` command to find the name of your routine.
- Use the source level debugging support to locate your routine.

Rebuilding the Application

If you can rebuild the application that you are debugging, it is often easiest to compile a `DbgBreak` call into the problem routine. Palm Debugger will break on the line containing that call.

Setting an A-trap Break

If you know that the problem routine makes a certain system call, you can use debugging commands to set an a-trap break on that call. The potential problem with this method is that other routines might make the same system call, which means that you will get false triggers.

For example, if you want to find your application's main event loop, you can use the following steps.

1. Set an a-trap break for the `EvtGetEvent` system call, and then tell Palm Debugger to go until it hits a break, as shown here:

```
atb "evtgetevent"
A-trap set on 011d (evtgetevent)
g
Remote stopped due to: A-TRAP BREAK EXCEPTION
'EvtGetEvent'
```

```
+$0000 10C3B1E2 *LINK A6,$0000 | 4E56 0000
```

When Palm Debugger breaks due to an a-trap break, the current location is at the beginning of the system call. This means that the return address on the stack is the function that made the system call. In the above example, this will be your application's main event loop.

2. Set a temporary breakpoint at the function return address that is currently on the stack. You can use the @ operator to fetch the long word at the stack pointer, as shown here:

```
gt @sp
EXCEPTION ID = $80
'EventLoop'
+$0016 1001B2E6 *MOVE.L A2,-(A7) | 2F0A
```

The program counter is now at the instruction in your main event loop that immediately follows the `EvtGetEvent` call.

3. Disassemble your main event loop. You can use the colon (`:`) symbol to easily grab the starting address of the current routine.

```
il :
'EventLoop 1001B2D0'
+$0000 1001B2D0  LINK A6,-$001C | 4E56 FFE4
+$0004 1001B2D4  MOVEM.L D3-D4/A2,-(A7) | 48E7
1820
+$0008 1001B2D8  LEA -$0018(A6),A2 | 45EE FFE8
+$000C 1001B2DC  PEA $00000032 ; 00000032 |
4878 0032
+$0010 1001B2E0  MOVE.L A2,-(A7) | 2F0A
+$0012 1001B2E2  _EvtGetEvent ; $10C3B1E2 |
4E4F A11D
+$0016 1001B2E6 *MOVE.L A2,-(A7) | 2F0A
+$0018 1001B2E8  _SysHandleEvent ; $10C0E9EC |
4E4F A0A9
+$001C 1001B2EC  ADD.W #$000C,A7 | DEFC 000C
+$0020 1001B2F0  TST.B D0 | 4A00
```

The `atb`, `g`, `gt`, and `il` commands are described in detail in Chapter 3, "Palm Debugger Command Reference."

Using the Find Text Command

Another method for finding a certain code routine is to search through memory for the name of the routine. You can use Palm Debugger's `ft` command to search for text. This command takes three arguments: the text to find, the starting address of the search, and the number of bytes to search.

For example, to search through the first megabyte of RAM on a Palm III, you can use the following command:

```
ft "EventLoop" 10000000 100000
dm 100005C4 ;100005C4: 45 76 65 6E 74 4C 6F 6F
70 63 61 74 69 6F 6E 00 "EventLoop......"
```

NOTE: RAM starts at address `0x10000000` in all current Palm handheld devices except for the Palm V. RAM starts at address `0` on the Palm V.

To search ROM instead, use address `0x10C00000`.

You can repeat the find, starting from the current location, by pressing the Enter key.

```
dm 1001B355 ;1001B355: 45 76 65 6E 74 4C 6F 6F
70 00 00 4E 56 00 00 2F "EventLoop..NV../"
```

Again, you can ensure that the routine you've found is the one you want, you can disassemble the current routine by entering the following command:

```
il :
```

IMPORTANT: In the above example, the `ft` command first found the text at address `0x100005C4`. This is actually a a copy of the search string the debugger nub is using. You must search a second time to find the first "actual" instance of the text string.

The `ft` and `il` commands are described in detail in Chapter 3, "Palm Debugger Command Reference."

Using the Source Level Debugging Support

If you have built your application with the gcc compiler and generated a symbol file, you can find your code by following these steps:

1. Launch the console nub on the handheld device, as described in Activating Console Input.
2. Open your symbols file. You can use the **Open Symbol File** command from Palm Debugger's Source menu.
3. After the symbol file has loaded, choose the **Break** command from the Source menu to break into the debugger nub on the device.
4. In the source window, select the source line of the routine you want to debug.
5. Choose the **Toggle Breakpoint** command from the Source menu to set the breakpoint.

Finding Memory Corruption Problems

As anyone who has tried knows, finding the routine that is trashing memory can be a very frustrating task. A memory bug can trash the low memory globals used by the system, the dynamic memory heap, or an application variable, any of which can cause unpredictable behavior. This section provides tips for tracking down two kinds of memory bugs:

- heap corruptions
- application variable corruption

Tracking Down Heap Corruption

If you suspect a corrupted heap, check the heap. You can perform a fast check of the heap with the `hchk` command, which verifies the validity of the heap. For example:

```
hchk 0
Heap OK
```

You can also use the `hd 0` command to display a dump of the dynamic heap. If the heap is in a valid state, the heap dump will complete and you will see the heap summary displayed at the bottom of the window. For example:

```
hd 0

Displaying Heap ID: 0000, mapped to 00001480
                    req   act
resType/  #resID/
 start    handle   localID   size   size  lck own
flags type  index attr ctg  uniqueID name
--------------------------------------------------
------------------------------------------
-00001534 00001494 F0001495 000456 00045E  #0  #0
fM Graffiti Private
-00001992 00001498 F0001499 000012 00001A  #0  #0
fM DataMgr Protect List (DmProtectEntryPtr*)
-000019AC 00001490 F0001491 00001E 000026  #0  #0
fM Alarm Table
-000019D2 0000148C F000148D 000038 000040  #0  #0
fM
*00001A12 0000149C F000149D 000396 00039E  #2  #1
fM Form "3:03 pm"
*00001DB0 000014A0 F00014A1 00049A 0004A2  #2  #0
fM
 00002252 -------- F0002252 00002E 00003E  #0  #0
FM
 00002290 -------- F0002290 00EC40 00EC50  #0  #0
FM
-00010EE0 -------- F0010EE0 000600 000608  #0 #15
fM Stack: Console Task

...

000114E8 -------- F00114E8 000FF8 001008  #0  #0
FM
-000124F0 -------- F00124F0 001000 001008  #0 #15
fM
-00017D30 -------- F0017D30 00003C 000044  #0 #15
fM SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15
fM Feature Manager Globals (FtrGlobalsType)
```

```
-00017D84 -------- F0017D84 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Update 3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15
fM DmOpenRef: 'Update 3.0.2'
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15
fM Handle Table: 'Ô©Update 3.0.2'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Ô©Update 3.0.2'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15
fM DmOpenRef: 'Ô©Update 3.0.2'
--------------------------------------------------
----------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010C50 bytes)
  Movable Chunks:     #51     (005E80 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

If you break into the debugger nub at various points during the execution of your application and check the heap, you can narrow down where the corruption is occurring in your code.

Another method for tracking down heap corruption is to use the `mdebug` command, which puts the handheld device into one of several heap checking modes. Once a heap-checking mode has been activated on the device, the Palm OS performs an automatic heap check and verification after each call to the Memory Manager. If the heap is corrupted, the system automatically breaks into the debugger. The following is an example of the mdebug command:

```
mdebug -partial
Current mode = 001A
Only Affected heap checked/scrambled per call
Heap(s) checked on EVERY Mem call
Heap(s) scrambled on EVERY Mem call
Free chunk contents filled & checked

Minimum dynamic heap free space recording OFF
```

Note that the memory checking modes can seriously degenerate the performance of an application. You can enable or disable various `mdebug` options to strike a balance between performance and debugging information. For more information, see MDebug.

The `hd`, `hchk`, and `mdebug` commands are described in detail in Chapter 3, "Palm Debugger Command Reference."

Tracking Down Global Variable Corruption

When you have a bug that is trashing a system or application global, you must first determine which address in memory is being corrupted. Once you know that address, you can use the Step-Spy (`ss`) command to watch the address. The `ss` command puts the processor into single-step mode and automatically checks the contents of a specified address after each instruction. If the instruction causes the contents of the address the change, the debugger breaks. For example:

```
ss 100
Step Spying on address: 00000100
```

Note that the `ss` command is single-stepping through instructions, and thus the handheld device runs slowly. Ideally, you can narrow down the range of code involved with the corruption and use this command to watch the execution of this code section.

Displaying Local Variables and Function Parameters

If you are debugging with the source window, the current function's local variables and parameters are displayed in the upper pane of the window. However, if you do not have access to symbol information, you need to use debugging commands to manually look up the variable values. This section describes the steps you need to take to look up values for a typical function, which is shown in Listing 2.4

Listing 2.4 An example function for viewing local variables and parameters

```
static Boolean
MainFrmEventHandler (EventPtr eventP)
```

```
{
  FormPtr   formP;
  Boolean   handled = false;
  Err       err;
  char      buffer[64];
  UInt32    numBytes=0;
  Int16     i;
  static    char prevChar = 0;

  // See if StdIO can handle it
  if (StdHandleEvent (eventP)) return true;

  // body of function omitted for clarity
  ...

  return false;
}
```

If you break into the debugger and disassemble the code at the beginning of this function, just before it calls the `StdHandleEvent` function, this is what you see:

```
il :
'MainFrmEventHandler 1001E296'
+$0000 1001E296  LINK A6,-$0048 | 4E56 FFB8
+$0004 1001E29A  MOVEM.L D3-D5/A2,-(A7) | 48E7
1C20
+$0008 1001E29E  MOVE.L $0008(A6),A2 | 246E
0008
+$000C 1001E2A2  CLR.B D5 | 4205
+$000E 1001E2A4  CLR.L -$0044(A6) | 42AE FFBC
+$0012 1001E2A8 *MOVE.L A2,-(A7) | 2F0A
+$0014 1001E2AA  BSR.W StdHandleEvent ;
1001F214        | 6100 0F68
+$0018 1001E2AE  ADDQ.W #$04,A7 | 584F
+$001A 1001E2B0  TST.B D0 | 4A00
+$001C 1001E2B2  BEQ.S
MainFrmEventHandler+$0024 ; 1001E2BA | 6706
```

The first `UInt32` on the stack upon function entry is the return address for the function. Immediately following that are the parameter values, from left to right. In the listing above, if you display the memory pointed to by the stack pointer at the beginning of the function, you see the following:

```
dm sp
00014A2A: 10 C4 77 00 00 01 4A 4E 00 01 4A 4E
00 01 51 0E "..w...JN..JN..Q."
```

The first `UInt32` (`0x10C47700`) is the return address of the function.

The second `UInt32` (`0x00014A4E`) is the value of the function's `eventP` parameter.

After the LINK instruction executes however, the stack pointer register is changed: the stack pointer is decremented to make room for a saved value of the `A6` register and for local variables; in this example, there are `0x48` bytes of local variables.

After the LINK instruction executes, the `A6` register is changed to point to the beginning of the functions' stack frame. This register is used by the function to access parameters and local variables. The following shows what the stack looks like after the LINK instruction executes:

```
Address : Contents
  ------------------------------------------------
-------
  A7 => 149CE                   <= new "top" of
stack
              : ...             <= 0x48 bytes of
local variables
  A6 => 14A26 : 00 01 4A 3A     <= saved value of
A6
    14A2A : 10 C4 77 00    <= return address
    14A2E : 00 01 4A 4E<= eventP parameter
```

If you display the memory referenced by register `A6` at this time, you see the following:

```
dm a6
00014A26: 00 01 4A 3A 10 C4 77 00 00 01 4A 4E
00 01 4A 4E "..J:..w...JN..JN"
```

The first `UInt32` pointed to by A6 is the old value of `A6`, the next `UInt32` is the return address of the routine, and following that are the function parameter values. This means that the first parameter to the function can always be found at `8(A6)`.

Any local variables belonging to the function are stored in memory locations preceding `A6`. In the above example, the `numBytes` local variable is located at `-$0044(A6)`. Once you know the offset of the variable, you can access by using an offset from the `A6` register; thus, you can use the following command to view the `numBytes` parameter:

```
dm -44+a6
000149E2: 00 00 00 00 00 00 1A 0C 20 00 20 04
00 01 4A 08 "........ . ...J."
```

Changing the Baud Rate Used by Palm Debugger

Both the debugger and console nubs on the handheld device always start communicating at 57,600 baud. You can change this baud rate by selecting a new speed from Palm Debugger's Communications menu.

If you are using a serial cable that does not include hardware handshaking lines, you might need to switch to a lower baud rate. And if you are downloading a large file to the handheld device, you might want to switch higher baud rate. Palm Debugger allows you to set the baud rate to values ranging from 2400 baud to 230,400 baud.

When you choose a new baud rate, Palm Debugger sends a request packet to the nub on the handheld device to change its baud rate, and then Palm Debugger changes its own baud rate. If Palm Debugger is attached to the debugger nub on the device, the request goes to the debugger nub; otherwise, the request goes to the console nub.

In either case, changing the baud rate of either nub on the handheld device changes the baud rate of both nubs.

NOTE: The new baud rate is only in effect until you soft reset the handheld device.

Debugging Applications That Use the Serial Port

Although it is very difficult to debug an application that uses the serial port, you can still use a limited set of debugging functions. You cannot use the console nub while an application on the handheld device is using the serial port.

When you do enter the debugger nub on the handheld device while debugging a serial application, the debugger sends data over the serial port and probably disrupts the application’s communications. At that point, you can switch the serial cable back over to Palm Debugger, double-check your baud rate setting, attach to the device with the `att` command, and perform “post-mortem” analysis of the problem.

Making Sure the Baud Rates Match

If the debugger nub on the handheld device has already been entered at least once, and you later launch a handheld application that opens the serial port, that application might change the port speed. The debugger nub will then use the new baud rate, but you will need to manually change the baud rate that Palm Debugger is using for communications to work. Use Palm Debugger’s Communications menu to change the speed.

Importing System Extensions and Libraries

You can use the console window `import` command to copy a new database or replace an existing database on the handheld device. However, the `import` command cannot replace a database that is currently opened.

If you are developing a system extension or shared library and need to use the `import` command, you need to do some extra work. This is due to the fact that system extension databases and shared libraries are generally either opened or marked as protected. To import a newer version of a system extension database or shared library, you have to make sure that the old database has been closed and is not protected; otherwise, the `import` command generates the following message:

```
###Error $00000219 occurred
```

To get around this problem, you need to perform a soft reset on the handheld device and tell the Palm OS to not automatically load system extensions or shared libraries. To do so, follow these steps:

1. Press the Up button on the handheld device while pressing the reset button on the back of the device with a paper clip or similar blunt object. This tells the Palm OS on the device to not load the system extension databases and shared libraries.
2. Start the console nub on the handheld device.
3. Import your system extension or shared library with the `import` command.
4. Perform another soft reset on the device, and the system will use the new version of the extension or library.

Determining the Current Location Within an Application

You can use one of the following three methods to determine where you are in your code:

1. Disassemble code starting at the beginning of the current routine, using the following command:

```
il :
'EventLoop 1001B2D0'
+$0000 1001B2D0  LINK A6,-$001C | 4E56 FFE4
+$0004 1001B2D4  MOVEM.L D3-D4/A2,-(A7) | 48E7
1820
+$0008 1001B2D8  LEA -$0018(A6),A2 | 45EE FFE8
+$000C 1001B2DC  PEA $00000032 ; 00000032 |
4878 0032
+$0010 1001B2E0  MOVE.L A2,-(A7) | 2F0A
+$0012 1001B2E2  _EvtGetEvent ; $10C3B1E2 |
4E4F A11D
+$0016 1001B2E6 *MOVE.L A2,-(A7) | 2F0A
```

```
+$0018 1001B2E8  _SysHandleEvent ; $10C0E9EC |
4E4F A0A9
+$001C 1001B2EC  ADD.W #$000C,A7 | DEFC 000C
+$0020 1001B2F0  TST.B D0 | 4A00
```

2. Perform a stack crawl with the `sc` command, which displays the oldest routine at the top and the newest at the bottom. For example:

```
sc
Calling chain using A6 Links:
A6 Frame Caller
00000000 10C68982 cjtkend+0000
00015086 10C6CA26 __Startup__+0060
00015066 10C6CCCE PilotMain+0250
00014FC2 10C0F808 SysAppLaunch+0458
00014F6E 10C10258 PrvCallWithNewStack+0016
0001491E 1001CC7E start+006E
000148E6 1001CF44 PilotMain+001C
```

3. Get a list of the currently opened databases. Your application should be one of the listed databases. Note that the `System` and `GraffitiShortCuts` databases are always opened by the system, and will appear at the bottom of the list. Use the `opened` command as follows:

```
opened

name             resDB cardNum accessP        ID
openCnt mode
--------------------------------------------------
----------------------------
LauncherDB       no      0     000151460001814F  1
  0003
*Launcher        yes     0     00016DD200D1FA98  1
  0001
*Graffiti ShortCutsyes  0     00017D5C001FFE7F  1
  0007
*System          yes     0     00017FEE00D20A44  1
  0005
```

```
--------------------------------------------------
----------------------------
Total: 4 databases opened
```

3

Palm Debugger Command Reference

This chapter describes Palm Debugger commands. For an introduction to using Palm Debugger, see Chapter 2, "Using Palm Debugger."

This chapter begins with a description of the syntax used to describe commands, and then expands into the following sections:

- Debugging Window Commands provides a reference description for each command that you can use in the debugging window to communicate with the debugger nub running on the handheld device. The command reference listings are ordered alphabetically.
- Debugging Command Summary provides tables that summarize the debugging commands by category.

Command Syntax

This chapter uses the following syntax to specify the format of debugger commands:

```
commandName <parameter> [options]
```

`commandName`	The name of the command.
`parameter`	Parameter(s) for the command. Each parameter name is enclosed in angle brackets (< and >). Sometimes a parameter can be one value or another. In this case the parameter names are bracketed by parentheses and separated by the **|** character.

`options`	Optional flags that you can specify with the command. Note that options are specified with the dash (-) character in the console window and with the backslash (\) character in the debugging window.

NOTE: Any portion of a command that is shown enclosed in square brackets ("[" and "]") is optional.

The following is an example of a command definition

```
dir (<cardNum>|<srchOptions>) [displayOptions]
```

The `dir` command takes either a card number of a search specification, followed by display options.

Here are two examples of the `dir` command sent from the console window:

```
dir 0 -a
dir -t rsrc
```

And here are the same two commands sent from the debugging window:

```
dir 0 \a
dir \t rsrc
```

Specifying Command Options

All command options and some command parameters are specified as flags that begin with a dash (in the console window) or backslash (in the debugging window). For example:

```
-c
-enable
\enable
```

Some flags are followed by a keyword or value. You must leave white space between the flag and the value. For example:

```
-f D:\temp\myLogFile
\t Rsrc
```

Specifying Numeric and Address Values

Many of the debugging commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (#) or binary (%). Table 3.1 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 3.1 Specifying numeric values in Palm Debugger

Hex value	Decimal value	Binary value
`64 or $64`	`#100`	`%01100100`
`F5 or $F5`	`#245`	`%11110101`
`100 or $100`	`#256`	`%100000000`

For more information, see Specifying Constants.

Using the Expression Language

When you send commands from the debugger window to the debugger nub on the handheld device, you can use Palm Debugger's expression language to specify the command arguments. This language is described in Using Debugger Expressions.

Debugging Window Commands

You use Palm Debugger's debugging window to send commands to the debugger nub that is running on the handheld device.

NOTE: You can use Palm Debugger's expression language to specify arguments to debugging window commands. The expression language is described in Using Debugger Expressions.

This section provides a description of all of the commands in alphabetical order. For convenience, the commands are categorized here:

Table 3.2 Debugging window command categories

Category	Commands
Console	cardInfo, dir, hChk, hd, hl, ht, Info, opened, storeInfo
Flow Control	att, atb, atc, atd, br, brc, brd, cl, dx, g, gt, s, ss t, reset
Memory	atr, db, dl, dm, dw, fb, fill, fl, ft, fw, il, sb, sc, sc6, sc7, sl, sw, wh
Miscellaneous	help (?), penv
Register	reg
Template	>, sizeof, templates, typedef, typeend
Utility	alias, aliases, bootstrap, keywords, load, run, save, , var, variables

>

Purpose Defines a structure field.

Usage `> <typeName> <"fieldName">`

Parameters

`typeName`	The type of the field.
`fieldName`	The quoted name of the field in the template.

Comments Use the > command in conjunction with the typedef and typeend commands to defined structure templates that you can use to display complex structures with a single memory display (dm) command.

Example

```
typedef struct "PointType"
> SWord "X"
> SWord "Y"
typeend
```

alias

Purpose Defines or displays an alias.

Usage
```
alias <"name">
alias <"name"> <"definition">
```

Parameters

`name`	The quoted name of the alias.
`definition`	The quoted definitional text for the alias.

Comments Use the `alias` command to define an alias for a command or group of commands.

If you provide only the name of an alias, this command displays the definition for that name.

Example
```
alias "ls" "dir"
```

aliases

Purpose Displays the names of all defined aliases.

Usage `aliases`

Parameters None.

Comments

Example
```
aliases
ls
```

atb

Purpose Adds an A-Trap break.

Usage

```
atb (<"funcName"> | <trapNum>)
([libRefNum> | <"libName">])
```

Parameters

`funcName`	The quoted name of the function.
`trapNum`	The A-Trap number.
`libRefNum`	Optional. the reference number for the library in which the function resides.
`libName`	Optional. The quoted name of the library in which the function resides.

atc

Purpose Clears an A-Trap break.

Usage

```
atc (<"funcName"> | <trapNum>)
([libRefNum> | <"libName">])
```

Parameters

`funcName`	The quoted name of the function.
`trapNum`	The A-Trap number.
`libRefNum`	Optional. the reference number for the library in which the function resides.
`libName`	Optional. The quoted name of the library in which the function resides.

atd

Purpose Displays a list of all the A-Trap breaks currently set.

Usage

```
atd
```

Parameters None.

atr

Purpose Registers a function name with an A-Trap number.

Usage `atr <"funcName"> <trapNum> [<"libName">]`

Parameters

`funcName`	The quoted name of the function.
`trapNum`	The A-Trap number.
`libName`	Optional. The quoted name of the library in which the function resides.

att

Purpose Attach to the handheld device.

Usage `att [options]`

Parameters

`options`	You can optionally specify the following options: `\async` Attach asynchronously.

Example

```
att
EXCEPTION ID = $A
  +$0512  10C0EEFE  *MOVEQ.L   #$01,D0 | 7001
```

bootstrap

Purpose Loads a ROM image into memory on the handheld device, using the bootstrap mode of the Dragonball EZ processor.

Usage `bootstrap <"hwInitFileName"> <"romFileName"> [options]`

Parameters

`hwInitFileName`	The quoted name of the hardware initialization file on your desktop computer.
`romFileName`	The quoted name of the ROM image file on your desktop computer.
`options`	You can optionally specify the following options: `\slow` ???.

br

Purpose Sets a breakpoint at the specified address.

Usage `br [options] <addr>`

Parameters

`options`	Optional. You can specify the following option: `\toggle` Toggles the breakpoint on or off.
`addr`	The memory address at which to set the breakpoint.

brc

Purpose Clears a breakpoint or all breakpoints.

Usage

```
brc
brc <addr>
```

Parameters `addr` A memory address.

Comments Use the `br` command to clear a specific breakpoint or to clear all breakpoints. if you specify a valid address value, that breakpoint is cleared. If you do not specify any address value, all breakpoints are cleared.

NOTE: The `cl` and `brc` commands are identical.

brd

Purpose Displays a list of all of the breakpoints that are currently set.

Usage

```
brd
```

Parameters None.

cardInfo

Purpose Retrieves information about a memory card.

Usage

```
cardinfo <cardNum>
```

Parameters `cardNum` The number of the card for which you want information displayed. You almost always use `0` to specify the built-in RAM.

Comments

NOTE: You can use the `cardinfo` command in either the Console window or the Debugger window.

Example

```
cardinfo 0

Name: PalmCard
Manuf: Palm Computing
Version: 0001
CreationDate: B1243780
ROM Size: 00118FFC
RAM Size: 00200000
Free Bytes : 0015ACB2
Number of heaps: #3
```

cl

Purpose Clears a breakpoint or all breakpoints.

Usage

```
cl
cl <addr>
```

Parameters

addr	A memory address.

Comments Use the `cl` command to clear a specific breakpoint or to clear all breakpoints. if you specify a valid address value, that breakpoint is cleared. If you do not specify any address value, all breakpoints are cleared.

NOTE: The `cl` and `brc` commands are identical.

db

Purpose Displays the byte value at a specified address.

Usage

```
db <addr>
```

Parameters

addr	A memory address.

Comments

Example

```
db 0100
Byte at 00000100 = $01  #1  #1  '.'
```

dir

Purpose Displays a list of the databases on the handheld device.

Usage `dir (<cardNum>|<searchOptions>) [<displayOptions>]`

Parameters

`cardNum`	The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.
`searchOptions`	Optional. Options for listing a specific database. Specify any combination of the following flags. `\c <creatorID>` Search for a database by creator ID. `\latest` List only the latest version of each database. `\t <typeID>` Search for a database by its type.
`displayOptions`	Optional. Options for which information is displayed in the listing. Specify any combination of the following flags. `\a` Show all information. `\at` Show the database attributes. `\d` Show the database creation, modification, and backup dates. `\i` Show the database appInfo and sortInfo field values. `\id` Show the database chunk ID `\s` Show the database size `\m` Show the database modification number.

`\n`	Show the database name.
`\r`	Show the number of records in the database.
`\tc`	Show the database type ID and creator ID.
`\v`	Show the database version number.

Comments Use the `dir` command to display a list of the databases on a specific card or in the handheld device built-in RAM. You typically use the following command to list all of the databases stored in RAM on the handheld device:

```
dir 0
```

Or use the -a switch to display all of the information for each database:

```
dir 0 -a
```

NOTE: You can use the `dir` command in either the Console window or the Debugger window. However, the command options must be prefaced with the "\" character in the debugger window, rather than with the "-" character that you use in the console window version.

Example

```
dir 0

name                        ID      total       data
--------------------------------------------------------
*System              00D20A44  392.691 Kb  390.361
Kb
*AMX                 00D209C4   20.275 Kb   20.123
Kb
*UIAppShell          00D20944    1.327 Kb    1.175
Kb
*PADHTAL Library     00D208E2    7.772 Kb    7.674
Kb
*IrDA Library        00D20876   39.518 Kb   39.402
Kb
 ...
```

```
 MailDB             0001817F    1.033 Kb    0.929
Kb
 NetworkDB          0001818B    0.986 Kb    0.722
Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842
Kb
 DatebookDB         000181FB    0.084 Kb    0.000
Kb
--------------------------------------------------
-
Total: 41
```

dl

Purpose Displays the 32-bit long value at a specified address.

Usage `dl <addr>`

Parameters

`addr`	A memory address.

Example

```
  dl 0100
Long at 00000100 = $01010000  #16842752  #16842752
'....'
```

dm

Purpose Displays memory for a specified number of bytes or templates.

Usage `dm <addr> [<count>] [<template>]`

Parameters

`addr`	A memory address.
`count`	Optional. The number of bytes to display.
`template`.	The name of the structure template to use. This defines how much memory to display and how to display it.

Comments Use the `dm` command to display a range of memory values. You can specify a byte count or a structure template; if you do not specify either, `dm` displays sixteen bytes of memory.

Example

```
dm 0100 8
00000100: 01 01 00 00 02 B0 00 01
```

dump

Purpose Dumps memory to a file.

Usage `dump <“filename”> <addr> <numBytes>`

Parameters

`filename`	The quoted name of the file to which the data is to be written.
`addr`	A memory address.
`numBytes`	The number of bytes of memory to write to the file.

Comments Use the `dump` command to write a dump of a range of memory addresses to file.

dw

Purpose Displays the 16-bit word value at a specified address.

Usage `dw <addr>`

Parameters

`addr`	A memory address.

Example

```
dw 0100
Word at 00000100 = $0101  #257  #257  '..'
```

dx

Purpose Enables or disables `DbgBreak()` breaks.

Usage `dx`

Parameters None.

fb

Purpose Searches through a range of memory for a specified byte value.

Usage `fb <value> <addr> <numBytes> [flags]`

Parameters

`value`	The byte value to find.
`addr`	The address at which to start the seearch.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags: `\a` Find all occurrences within the specified range. `\i` Use caseless comparison.

Comments By default, `fb` uses a case sensitive comparison.

Example

```
  fb ff 0100 200

dm 00000110   ;00000110: FF 00 00 00 03 18 00 00
03 BC 00 01 7D 72 00 01 "............}r.."
```

fill

Purpose Fills memory with a specified byte value.

Usage `fill <addr> <numBytes> <value>`

Parameters

`addr`	A memory address.
`numBytes`	The number of bytes to fill with the value.
`value`	The value assigned to each byte.

Example

```
fill 0100 8 FF
```

fl

Purpose Searches through a range of memory for a specified 32-bit long value.

Usage `fb <value> <addr> <numBytes> [flags]`

Parameters

`value`	The byte value to find.
`addr`	The address at which to start the search.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags:
	`\a` Find all occurrences within the specified range.
	`\i` Use caseless comparison.

Comments By default, `fl` uses a case sensitive comparison.

Example

```
fl ffff 0 1000
```

```
dm 00000034   ;00000034: FF FF 00 00 FF FF 00 00
FF FF 00 00 FF FF 00 00      "................"
```

ft

Purpose Searches through a range of memory for the specified text.

Usage `ft <text> <addr> <numBytes> [flags]`

Parameters

`text`	The quoted text to find.
`addr`	The address at which to start the seearch.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags: `\a` Find all occurrences within the specified range. `\i` Use caseless comparison.

Comments By default, `ft` uses a case sensitive comparison.

Example

```
ft "abc" 0 1000
```

```
dm 000005C4   ;000005C4: 61 62 63 27 00 00 00 00
00 01 4B 06 00 00 0
```

fw

Purpose Searches through a range of memory for the specified 16-bit word value.

Usage `fw <value> <addr> <numBytes> [flags]`

Parameters

`value`	The value to find.
`addr`	The address at which to start the seearch.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags: `\a` Find all occurrences within the specified range.

`\i`	Use caseless comparison.

Comments By default, `fw` uses a case sensitive comparison.

Example

```
fw 32000 0 1000

dm 00000258   ;00000258: 00 20 00 00 00 07 A7 0E
00 00 00 01 00 00 00 00      ". ..............."
```

g

Purpose Continues execution.

Usage

```
g
g &lt;addr&gt;
```

Parameters

`addr`	Optional. The address from which to continue execution.

Comments You can optionally specify a starting address for the `g` command. If you do not specify an address, execution continues from the current program counter location.

Example

```
g
```

gt

Purpose Sets a temporary breakpoint at the specified address, and resumes execution from the current program counter.

Usage

```
gt <addr>
```

Parameters

`addr`	The address at which to set the breakpoint. If you do not specify an address, the current program counter location is used.

Comments

hChk

Purpose Checks the integrity of a heap.

Usage `hchk <heapId> [options]`

Parameters

`heapId`	The hexadecimal number of the heap whose contents are to be checked. Heap number `0x0000` is always the dynamic heap.
`options`	Optional. You can specify the following option: `\c` Check the contents of each chunk.

Comments

NOTE: You can use the `hchk` command in either the Console window or the Debugger window. However, the command options must be prefaced with the "\" character in the debugger window, rather than with the "-" character that you use in the console window version.

Example

```
hchk 0000
Heap OK
```

hd

Purpose Displays a hexadecimal dump of the specified heap.

Usage `hd <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap whose contents are to be displayed. Heap number `0x0000` is always the dynamic heap.

Comments Use the hd command to display a dump of the contents of a specific heap from the handheld device. You can use the HL command to display the heap IDs.

Example

```
hd 0

Displaying Heap ID: 0000, mapped to 00001480
                    req   act
resType/  #resID/
 start    handle   localID   size   size  lck own
flags type  index attr ctg  uniqueID name
--------------------------------------------------
------------------------------------------
-00001534 00001494 F0001495 000456 00045E  #0  #0
fM Graffiti Private
-00001992 00001498 F0001499 000012 00001A  #0  #0
fM DataMgr Protect List (DmProtectEntryPtr*)
-000019AC 00001490 F0001491 00001E 000026  #0  #0
fM Alarm Table
-000019D2 0000148C F000148D 000038 000040  #0  #0
fM
*00001A12 0000149C F000149D 000396 00039E  #2  #1
fM Form "3:03 pm"
*00001DB0 000014A0 F00014A1 00049A 0004A2  #2  #0
fM
 00002252 -------- F0002252 00002E 00003E  #0  #0
FM
 00002290 -------- F0002290 00EC40 00EC50  #0  #0
FM
-00010EE0 -------- F0010EE0 000600 000608  #0 #15
fM Stack: Console Task

...

000114E8 -------- F00114E8 000FF8 001008  #0  #0
FM
-000124F0 -------- F00124F0 001000 001008  #0 #15
fM
```

```
-00017D30 -------- F0017D30 00003C 000044  #0 #15
fM SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15
fM Feature Manager Globals (FtrGlobalsType)
-00017D84 -------- F0017D84 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Update 3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15
fM DmOpenRef: 'Update 3.0.2'
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15
fM Handle Table: 'Ô©Update 3.0.2'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Ô©Update 3.0.2'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15
fM DmOpenRef: 'Ô©Update 3.0.2'
--------------------------------------------------
------------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010C50 bytes)
  Movable Chunks:     #51     (005E80 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

help (?)

Purpose Displays a list of commands or help for a specific command.

Usage

```
help
help <command>
```

Parameters

`command`	The name of the command for which you want help displayed.

Comments

NOTE: You can use the `help` command in either the Console window or the Debugger window.

Example

```
help hchk

Do a Heap Check.
Syntax: hchk <hex heapID> [options...]
         -c    : Check contents of each chunk
```

hl

Purpose Displays a list of memory heaps.

Usage `hl <cardNum>`

Parameters

`cardNum` The card number on which the heaps are located. You almost always use `0` to specify the built-in RAM.

Comments Use the `hl` command to list the memory heaps in built-in RAM or on a card.

NOTE: You can use the `hl` command in either the Console window or the Debugger window.

Example

```
  hl 0

   index   heapID  heapPtr     size     free
maxFree    flags
--------------------------------------------------
----------
       0     0000 00001480 00016B80 00010C50
0000EC48     8000
       1     0001 1001810E 001E7EF2 0014AD6A
00147D3A     8000
```

```
      2      0002 10C08212 00118DEE 0000A01C
0000A014     8001
```

ht

Purpose Displays summary information for the specified heap.

Usage `ht 0`

Parameters None.

Comments The `ht` commands displays the summary information that is also shown at the end of a heap dump generated by the hd command.

> **NOTE:** You can use the `ht` command in either the Console window or the Debugger window.

Example

```
ht 0000
Displaying Heap ID: 0000, mapped to 00001480
--------------------------------------------
Heap Summary:
  flags:               8000
  size:                016B80
  numHandles:          #40
  Free Chunks:         #14      (010CAA bytes)
  Movable Chunks:      #48      (005E26 bytes)
  Non-Movable Chunks: #0       (000000 bytes)
```

il

Purpose Disassembles code in a specified line range.

Usage `il [<addr> | <“funcName”> [lineCount]]`

Parameters

`addr` Optional. The starting address at which to disassemble.

`funcName`	Optional. The name of the function whose code you want disassembled.
`lineCount`	Optional. If you provide a value for addr, you can also specify the number of lines of code to disassemble starting at `addr`.

Comments Use the `il` command to disassemble code. If you do not provide a function name or starting address value, disassembly begins at the current program counter value.

Example

```
  il 0100

          00000100   BTST      D0,D1             |
0101
          00000102   ORI.B     #$B0,D0 ; '.'
    | 0000 02B0
          00000106   ORI.B     #$30,D1 ; '0'
    | 0001 7830
          0000010A   ORI.B     #$01,D0 ; '.'
    | 0000 0001
          0000010E                              |
474A
          00000110   CoProc                      |
FF00 0000 0318
          00000116   ORI.B     #$BC,D0 ; '.'
    | 0000 03BC
          0000011A   ORI.B     #$72,D1 ; 'r'
    | 0001 7D72
          0000011E   ORI.B     #$BC,D1 ; '.'
    | 0001 6FBC
          00000122   ORI.B     #$22,D0 ; '"'
    | 0000 0722
```

Info

Purpose Displays information about a memory chunk.

Usage `info (<hexChunkPtr> | localID>) [options]`

Parameters

`hexChunkPtr` or `localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`options`
Optional. You can specify the following options:

`-card` <`cardNum`>
The card number if a local ID is specified instead of a chunk pointer.

Comments

NOTE: You can use the `info` command in either the Console window or the Debugger window. However, the command options must be prefaced with the "\" character in the debugger window, rather than with the "-" character that you use in the console window version.

Example

keywords

Purpose Lists all debugger keywords.

Usage `keywords`

Parameters None.

Example

```
keywords
```

```
t
g
SR
PC
SP
A7
A6
A5
A4
A3
A2
A1
A0
D7
...
```

load

Purpose Loads the data fork of a file at the specified address.

Usage `load <"fileName"> <addr>`

Parameters

`fileName`	The quoted name of the file whose data fork you want loaded.
`addr`	The memory address at which you want the data fork loaded.

opened

Purpose Lists all of the currently opened databases.

Usage `opened`

Parameters None.

NOTE: You can use the opened command in either the Console window or the Debugger window.

Example

```
opened

name                  resDB   cardNum   accessP
ID    openCnt         mode
-----------------------------------------------------
----------------------------
*Graffiti ShortCutsyes          0 00017D5C
001FFE7F         1       0007
*System          yes          0 00017FEE 00D20A44
1       0005
-----------------------------------------------------
----------------------------
Total: 2 databases opened
```

penv

Purpose Displays current environment information for the debugger.

Usage penv

Parameters None.

Comments The penv command displays the current values of the predefined debugger environment variables, which are summarized in Debugger Environment Variables.

Example

```
penv
============================
DebOut = false
SymbolsOn = true
StepRegs = false
ReadMemHack = false
```

```
Attached = true
............................
dot address = 00000000
last address = 00001022
last count = 0000000a
============================
```

reg

Purpose Displays all registers.

Usage `reg`

Parameters None.

Example

```
  reg

D0 = 00000102     A0 = 10C0EEF6     USP = BF6E446F
D1 = 00000013     A1 = 10C0EF0E     SSP = 000132E4
D2 = 00000027     A2 = 000133C2
D3 = 00000000     A3 = 00015404
D4 = 00014B06     A4 = 10CCFB7C
D5 = 00000000     A5 = 000149AA
D6 = 00D1EFE8     A6 = 000133AC     PC  = 10C0EEFE
D7 = 0001515E     A7 = 000132E4     SR  = tSxnzvc
Int = 0
```

reset

Purpose Performs a soft reset on the handheld device.

Usage `reset`

Parameters None.

Comments This command performs the same reset that is performed when you press the recessed reset button on a Palm Computing handheld device. It resets the memory system and reformats both cards.

> **NOTE:** You can use the `reset` command in either the Console window or the Debugger window.

Example

```
reset
Resetting system
```

run

Purpose Runs a debugger script from file.

Usage `run <"fileName">`

Parameters

`filename`	The quoted name of the file that contains the debugger script.

s

Purpose Single steps the processor, stepping into subroutines.

Usage `s`

Parameters None.

Example

```
s

'SysHandleEvent'
  +$0694  10C0F080  *MOVEM.L   (A7)+,D3-D5/A2-A4 |
4CDF 1C38
```

save

Purpose Saves a range of data from memory to file.

Usage `save <"fileName"> <addr> <numBytes>`

Parameters

`fileName`	The quoted name of the file to which you want the data saved.
`addr`	The starting address in memory to save.
`numBytes`	The number of bytes to save.

Example

```
save "savedMem1" 0100 100
```

sb

Purpose Sets the value of the byte at the specified address.

Usage `sb <addr> <value>`

Parameters

`addr`	The address of the byte.
`value`	The byte value.

Example

```
sb 0111 0a

Memory set starting at 00000111
```

sc

Purpose Displays a list of functions on the stack using information stored in the A6 frame pointer register.

Usage `sc [<addr> [<frames>]]`

Parameters

`addr`	Optional. The address from which to start listing.

`frames`	Optional. The number of frames to list. You can specify this only if you specify a value for `addr`.

Example

```
sc
Calling chain using A6 Links:
 A6 Frame   Caller
 00000000  10C68982  cjtkend+0000
 00015086  10C6CA26  __Startup__+0060
 00015066  10C6CCCE  PilotMain+0250
 00014FC2  10C0F808  SysAppLaunch+0458
 00014F6E  10C10258  PrvCallWithNewStack+0016
 00013414  10CCFBE0  __Startup__+0060
 000133F4  10CD08CE  PilotMain+0036
 000133DA  10CD6D18  EventLoop+0016
```

sc6

Purpose Lists the A6 stack frame chain, starting at the specified address.

Usage `sc6 [<addr> [<frames>]]`

Parameters

`addr`	Optional. The address from which to start listing.
`frames`	Optional. The number of frames to list. You can specify this only if you specify a value for `addr`.

Comments This command is the same as the sc command.

Example

```
sc
Calling chain using A6 Links:
 A6 Frame   Caller
 00000000  10C68982  cjtkend+0000
 00015086  10C6CA26  __Startup__+0060
 00015066  10C6CCCE  PilotMain+0250
 00014FC2  10C0F808  SysAppLaunch+0458
 00014F6E  10C10258  PrvCallWithNewStack+0016
 00013414  10CCFBE0  __Startup__+0060
```

```
000133F4  10CD08CE  PilotMain+0036
000133DA  10CD6D18  EventLoop+0016
```

sc7

Purpose Displays a list of functions on the stack using the stack pointer (A7). This displays information about functions on the stack that do not set up frame pointers

Usage `sc7 [<addr> [<frames>]]`

Parameters

`addr`	Optional. The address from which to start listing.
`frames`	Optional. The number of frames to list. You can specify this only if you specify a value for `addr`.

Comments Use the `sc7` command instead of the standard stack crawl command, `sc`, when you want to display information about routines on the stack that have not set up frame pointers. Note that this command will sometimes display bogus routines.

Example

```
  sc7

Return Addresses on the stack:
 Stack Addr   Caller
  00013AFC    00000000
  000133B0    10CD6D18  EventLoop+0016
  00013344    10C1F964
PrvHandleExchangeEvents+0028
```

sizeof

Purpose Displays the size, in bytes, of a template.

Usage `sizeof <template>`

Parameters

`template`	The name of the template.

Comments You can use the templates command to list the available templates.

Example

```
sizeof sdword
Size = 4 byte(s)
```

sl

Purpose Sets the value of the 32-bit long integer at the specified address.

Usage `sl <addr> <value>`

Parameters

`addr`	The address of the 32-bit value.
`value`	The long value.

Example

```
sl 0110 ffffffff
Memory set starting at 00000110
```

ss

Purpose Breaks into the debugger when the value of the long word at the specified address changes.

Usage `ss [<addr>]`

Parameters

`addr`	Optional. The address of the 32-bit value. If you do not specify an address value, the current program counter location is used. *

Example

```
ss 1000F024
```

storeInfo

Purpose Displays information about a memory store.

Usage `storeinfo <cardNum>`

Parameters

`cardNum`	The card number for which you want information displayed. You almost always use `0` to specify the built-in RAM.

Comments

NOTE: You can use the `storeinfo` command in either the Console window or the Debugger window.

Example

```
storeinfo 0

ROM Store:
  version: 0001
  flags: 0000
  name: ROM Store
  creation date: 00000000
  backup date: 00000000
  heap list offset: 00C08208
  init code offset1: 00C0D652
  init code offset2: 00C1471E
  database dirID: 00D20F7E

RAM Store:
  version: 0001
  flags: 0001
  name: RAM Store 0
  creation date: 00000000
```

```
backup date: 00000000
heap list offset: 00018100
init code offset1: 00000000
init code offset2: 00000000
database dirID: 0001811F
```

sw

Purpose Sets the value of the word at the specified address.

Usage `sw <addr> <value>`

Parameters

`addr`	The address of the 16-bit value.
`value`	The word value.

Example

```
sw 0110 ffff
Memory set starting at 00000110
```

t

Purpose Single steps the processor, stepping over subroutines.

Usage `t`

Parameters None.

Example

```
t

'SysHandleEvent'
Will Branch
  +$0514  10C0EF00  *BRA.W
SysHandleEvent+$0694 ; 10C0F080    |6000 017E
```

templates

Purpose Lists the names of the debugger templates.

Usage `templates`

Parameters None.

Example

```
templates

Char
Byte
SByte
Word
SWord
DWord
SDWord
```

typedef

Purpose Begins a structure definition block.

Usage `typedef struct <"name">`

Parameters

`name`	The quoted name of the template whose definition you are beginning.

Comments Use the `typedef` command in conjunction with the > and typeend commands to defined structure templates that you can use to display complex structures with a single memory display (dm) command.

Example

```
typedef struct "PointType"
> SWord "X"
> SWord "Y"
typeend
```

typeend

Purpose Ends a structure definition block.

Usage

```
typeend
```

Parameters None.

Comments Use the `typeend` command in conjunction with the > and typedef commands to defined structure templates that you can use to display complex structures with a single memory display (dm) command.

Example

```
typedef struct "PointType"
> SWord "X"
> SWord "Y"
typeend
```

var

Purpose Defines a debugger variable.

Usage

```
var <"name"> [<initialValue>]
```

Parameters

`name`	The quoted name of the variable that you are defining.
`initialValue`	Optional. The initial value for the variable. If you are assigning a string value to the variable, you must quote the initial value.

Example

```
var "testvar" 100

var "testvar" "Hello"
WARNING: redefining variable: testvar
```

variables

Purpose Lists the names of the debugger variables.

Usage `variables`

Parameters None.

Example

```
variables

DebOut
SymbolsOn
ReadMemHack
StepRegs
Attached
testvar
testvar2
```

wh

Purpose Displays system function information for a specified function name or A-Trap number. Also identifies the memory chunk that contains a specific address or lists all system functions.

Usage `wh [\a <addr>] [<"funcName"> | <ATrapNumber>]`

Parameters

`addr`	Specifies an address. The `wh` command displays the memory chunk that contains this address.
`funcName`	The quoted name of the system function for which you want information displayed.
`ATrapNumber`	The number of the A-trap number for which you want information displayed.

Debugging Command Summary

Flow Control Commands

`atb`	Adds an A-Trap break.
`atc`	Clears an A-Trap break.
`atd`	Displays a list of all A-Trap breaks.
`att`	Attach to the handheld device.
`br`	Sets a breakpoint at the specified address.
`brc`	Clears a breakpoint or all breakpoints.
`brd`	Displays a list of all breakpoints.
`cl`	Clears a breakpoint or all breakpoints.
`dx`	Enables or disables `DbgBreak()` breaks.
`g`	Continues execution.
`gt`	Sets a temporary breakpoint at the specified address, and resumes execution from the current program counter.
`reset`	Resets the memory system and formats both cards.
`s`	Single steps the processor, stepping into subroutines.
`ss`	Breaks into the debugger when the long word value at the specified address changes.
`t`	Single steps the processor, stepping over subroutines.

Memory Commands

`atr`	Registers a function name with an A-Trap number.
`db`	Displays the byte value at a specified address.

`dl`	Displays the 32-bit long value at a specified address.
`dm`	Displays memory for a specified number of bytes or templates.
`dw`	Displays the 16-bit word value at a specified address.
`fb`	Searches through a range of memory for a specified byte value.
`fill`	Fills memory with a specified byte value.
`fl`	Searches through a range of memory for a specified 32-bit long value.
`ft`	Searches through a range of memory for the specified text.
`fw`	Searches through a range of memory for the specified 16-bit word value.
`il`	Disassembles code in a specified line range.
`sb`	Sets the value of the byte at the specified address.
`sc`	Lists the A6 stack frame chain, starting at the specified address.
`sc6`	Lists the A6 stack frame chain, starting at the specified address.
`sc7`	Lists the A7 stack frame chain, starting at the specified address.
`sl`	Sets the value of the long at the specified address.
`sw`	Sets the value of the word at the specified address.
`wh`	Displays system function information for a specified function name or A-Trap number. Also identifies the memory chunk that contains a specific address or lists all system functions.

Template Commands

`>`	Defines a structure field.
`sizeof`	Displays the size, in bytes, of a template.
`templates`	Lists the names of the debugger templates.
`typedef`	Begins a structure definition block.
`typeend`	Ends a structure definition block.

Register Commands

`reg`	Displays all registers.

Utility Commands

`alias`	Defines or displays an alias.
`aliases`	Displays all debugger alias names.
`bootstrap`	Loads a ROM image into memory on the handheld device, using the bootstrap mode of the Dragonball EZ processor.
`flash`	Loads the file's data fork into Flash Memory at the specified address.
`keywords`	Lists all debugger keywords.
`load`	Loads the file's data fork at the specified remote address.
`run`	Runs a debugger script.
`save`	Saves a range of data from memory to file.
`var`	Defines a debugger variable.
`variables`	Lists the names of the debugger variables.

Console Commands

`CardInfo`	Retrieves information about a memory card.
`Dir`	Lists the databases.
`Dump`	Dumps a range of memory to a file.
`HChk`	Checks a heap.
`HD`	Displays a dump of a memory heap.
`HL`	Lists all of the memory heaps on the specified memory card.
`HT`	Performs a heap total.
`Info`	Displays information on a heap chunk.
`Opened`	Lists all currently opened databases.
`StoreInfo`	Retrieves information about a memory store.

Miscellaneous Debugger Commands

`help` (or `?`)	Displays a list of available commands.
help <cmd> or ? <cmd>	Displays help for a specific command.
`penv`	Displays debugger environment information.

Debugger Environment Variables

`DebOut`	A Boolean value that specifies if debug style output is enabled.
`ReadMemHack`	A Boolean value that specifies if the read memory hack is enabled.

`SymbolsOn`	A Boolean value that specifies if printing of disassembly symbols is enabled.
`StepRegs`	A Boolean value that specifies if register values should be shown after every step.

Predefined Constants

`true`	Integer value `1`.
`false`	Integer value `0`.
`srCmask`	The status register Carry bit.
`srImask`	The status register Interrupt field mask.
`srNmask`	The status register Negative bit.
`srSmask`	The status register Supervisor bit.
`srTmask`	The status register Trace bit.
`srVmask`	The status register Overflow bit.
`srXmask`	The status register extend bit.
`srZmask`	The status register Zero bit.

4

Using the Console Window

This chapter describes console window, which you can use with Palm® Debugger, the Palm Simulator, and the Metrowerks CodeWarrior environment to perform maintenance and high-level debugging of a Palm handheld device.

About the Console Window

The console window interfaces with a handheld device by sending information packets to and receiving information packets from the *console nub* on the device. The console interface provides a number of commands, which are used primarily for administration of databases and heap testing on handheld devices.

The console is available in three environments:

- as a separate window for sending and receiving commands in the Palm Debugger program, which is described in Chapter 2, "Using Palm Debugger."
- as a separate window that you can open from within the Palm Simulator program, which is described in Chapter 5, "Using the Palm Simulator."
- as a separate window that you can open within the Metrowerks CodeWarrior environment.

The console window provides the same commands and same interface in all three environments.

Before you use the console commands, you must connect your desktop computer with the console nub on the device, as described in Connecting the Console Window.

To learn more about using console commands, see the section Using the Console Window. For a complete reference description of each console command, see Console Window Commands. The commands are summarized in Console Command Summary.

Connecting the Console Window

Activating Console Input

To send console commands to the handheld device, you must connect your desktop computer to the handheld device, activate the console nub on the device, and then type commands into the console window.

The console nub runs as a background thread on the device, listening for commands on the serial or USB port. To activate the console nub, use the Graffiti Shortcut-2 command, as described in Using Shortcut Numbers to Activate the Windows.

When the console nub activates, it sends out a “Ready” message. If your desktop computer is connected to the device when the nub is activated, this message will display in the console window.

WARNING! The console nub activates at 57,600 baud, and your port configuration must match this is you are connecting over a serial port. You must set the connection parameters correctly for communications to work.

After you activate the console nub on the handheld device, the nub prevents other applications, including HotSync® from using the serial port. You have to soft-reset the handheld device before the port can be used.

Verifying Your Connection

To verify your device connection, you can type one of the simple console commands, such as `dir` or `hl 0`. If your connection is working and the console nub is active on the handheld device, you will see a list of memory heaps displayed in the window.

If the console nub is not running on the handheld device, or if the communications connection is not correctly configured, you will see an error message:

```
### Error $00000404 occurred
```

If you are certain that the console nub is running on the handheld, you need to set the connection parameters correctly. If you are using the console with Palm Debugger, you can use the Communications menu to set the parameters.

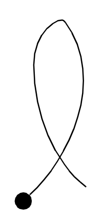
Using Shortcut Numbers to Activate the Windows

The Palm OS® responds to a number of "hidden" shortcuts for debugging your programs, including shortcuts for activating the console nub on the handheld device. You generate each of these shortcuts by drawing characters on your Palm Computing platform device, or by drawing them in the Palm OS Emulator emulator program, if you are using Palm OS Emulator to debug your program.

NOTE: If you open the Find dialog on the handheld device before entering a shortcut number, you get visual feedback as you draw the strokes.

To enter a shortcut number, follow these steps:

1. On your Palm Computing platform device, or in the emulator program, draw the shortcut symbol. This is a lowercase, cursive "L" character, drawn as follows:



2. Next, tap the stylus twice, to generate a dot (a period).

3. Next, draw a number character in the number entry portion of the device's text entry area. Table 4.1 shows the different shortcut numbers that you can use.

   For example, to activate the console nub on the handheld device, enter the follow sequence:

    .2

Table 4.1 Shortcut numbers for debugging

Number	Description	Notes
 .1	The device enters debugger mode, and waits for a low-level debugger to connect. A flashing square appears in the top left corner of the device.	This mode opens a serial port, which drains power over time. You must perform a soft reset or use the debugger's `reset` command to exit this mode.
 .2	The device enters console mode, and waits for communication, typically from a high-level debugger.	This mode opens a serial port, which drains power over time. You must perform a soft reset to exit this mode.
 .3	The device's automatic power-off feature is disabled.	You can still use the device's power button to power it on and off. Note that your batteries can drain quickly with automatic power-off disabled. You must perform a soft reset to exit this mode.

NOTE: These debugging shortcuts leave the device in a mode that requires a soft reset. To perform a soft reset, press the reset button on the back of the handheld with a blunt instrument, such as a paper clip.

Using the Console Window

You use the console window to enter console commands, which are typically used for administrative tasks such as managing databases on the handheld device. Commands that you type into the console window are sent to the console nub on the handheld device, and the results sent back from the device are displayed in the console window.

NOTE: Console command input is not case sensitive.

Table 4.2 shows the most commonly used console window commands.

Table 4.2 Commonly used console commands

Command	Description
`del`	Deletes a database from the handheld device.
`dir`	Displays a list of the databases on the handheld device.
`export`	Copies a Palm OS database from the handheld device to the desktop computer.
`import`	Copies a Palm OS database from the desktop computer to the handheld device.

Listing 4.1 shows an example of using console commands. In this example, **`boldface`** is used to denote commands that you type.

Listing 4.1 Importing a database into the handheld device

```
import 0 "C:Documents\MyDbs\Tex2HexApp.prc"

Creating Database on card 0
name: Text to Hex
type appl, creator TxHx

Importing resource 'code'=0....
```

```
Importing resource 'data'=0....
Importing resource 'pref'=0....
Importing resource 'rloc'=0....
Importing resource 'code'=1....
Importing resource 'tFRM'=1000....
Importing resource 'tver'=1....
Importing resource 'tAIB'=1000....
Importing resource 'Tbmp'=1000....
Importing resource 'Tbmp'=1001....
Importing resource 'MBAR'=1000....
Importing resource 'Talt'=1000....
Importing resource 'Talt'=1001....
Success!!

dir 0
name                    ID      total        data
--------------------------------------------------
*System            00D20A44  392.691 Kb  390.361
Kb
*AMX               00D209C4   20.275 Kb   20.123
Kb
*UIAppShell        00D20944    1.327 Kb    1.175
Kb
*PADHTAL Library   00D208E2    7.772 Kb    7.674
Kb
*IrDA Library      00D20876   39.518 Kb   39.402
Kb
*Net Library       00D207E2   86.968 Kb   86.780
Kb
*PPP NetIF         00D2073A   30.462 Kb   30.238
Kb
*SLIP NetIF        00D20692   15.812 Kb   15.588
Kb
*Loopback NetIF    00D20630    1.810 Kb    1.712
Kb
*MS-CHAP Support   00D205C4    4.342 Kb    4.226
Kb
*Network           00D203D2   40.442 Kb   39.624
Kb
```

```
*Address Book       00D20226   59.825 Kb   59.133
Kb
*Calculator         00D2002A   14.597 Kb   13.761
Kb
*Date Book          00D1FCF8  106.200 Kb  104.806
Kb
*Launcher           00D1FA98   36.633 Kb   35.617
Kb
*Memo Pad           00D1F91E   24.267 Kb   23.665
Kb
*Preferences        00D1F876    1.403 Kb    1.179
Kb
*Security           00D1F706    8.414 Kb    7.830
Kb
*HotSync            00D1F334   39.078 Kb   37.396
Kb
*To Do List         00D1F1E2   33.232 Kb   32.702
Kb
*Digitizer          00D1F126    2.002 Kb    1.742
Kb
*General            00D1EFE8    8.749 Kb    8.255
Kb
*Formats            00D1EF4A    4.732 Kb    4.526
Kb
*ShortCuts          00D1EE34    6.499 Kb    6.077
Kb
*Owner              00D1ED5A    4.095 Kb    3.781
Kb
*Buttons            00D1EC4E    7.419 Kb    7.015
Kb
*Modem              00D1EB74    8.222 Kb    7.908
Kb
*Mail               00D1E838   59.765 Kb   58.353
Kb
*Expense            00D1E614   42.304 Kb   41.396
Kb
*Unsaved Preferences  0001811B    0.898 Kb
0.550 Kb
*Net Prefs          00018133    0.084 Kb    0.000
Kb
```

```
 AddressDB         00018137   66.149 Kb   51.945
Kb
 MemoDB            0001815F    2.186 Kb    1.902
Kb
 ToDoDB            00018173    1.000 Kb    0.876
Kb
 MailDB            0001817F    1.033 Kb    0.929
Kb
 DatebookDB        000181EB   53.162 Kb   29.678
Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842
Kb
*Saved Preferences  00018123    3.753 Kb    3.031
Kb
 NetworkDB         0001818B    0.986 Kb    0.722
Kb
*Giraffe High Score  00018273    0.126 Kb    0.020
Kb
 Datebk3DB         0001827B    0.084 Kb    0.000
Kb
 ReDoDB            0001827F    0.084 Kb    0.000
Kb
 LauncherDB        0001814F    0.294 Kb    0.190
Kb
*MineHunt          00018287    9.810 Kb    9.264
Kb
*SubHunt           000182DF   17.700 Kb   16.758
Kb
*Puzzle            0001837F    5.256 Kb    4.886
Kb
*HardBall          000183B7   18.877 Kb   18.177
Kb
 Pictures          0001842B    0.084 Kb    0.000
Kb
*Jot               0001842F  120.409 Kb  119.841
Kb
*Graffiti ShortCuts    001FFE7F    2.872 Kb
2.766 Kb
*UnDupe            001FFE87    9.462 Kb    9.070
Kb
```

```
*WordView           001FFEC3   17.320 Kb   16.752
Kb
*SheetView          001FFF1F   56.753 Kb   55.877
Kb
 AOU Birds of NA    001FFE15  130.265 Kb   90.021
Kb
 ExpenseDB          001FBCB5    0.150 Kb    0.046
Kb
 DocsToGoDB         001FBCC1    0.326 Kb    0.202
Kb
 birds.PDB          001FBCD1    0.709 Kb    0.585
Kb
 foo                0001812F    0.084 Kb    0.000
Kb
*Text To Hex        001FFF85   34.725 Kb   33.827
Kb
---------------------------------------------------
--------------
Total: 59
```

These and all of the other console commands are described in detail in Console Window Commands.

Command Syntax

This chapter uses the following syntax to specify the format of debugger commands:

```
commandName <parameter> [options]
```

`commandName`	The name of the command.
`parameter`	Parameter(s) for the command. Each parameter name is enclosed in angle brackets (< and >). Sometimes a parameter can be one value or another. In this case the parameter names are bracketed by parentheses and separated by the **|** character.

`options`	Optional flags that you can specify with the command. Note that options are specified with the dash (-) character in the console window.

NOTE: Any portion of a command that is shown enclosed in square brackets ("[" and "]") is optional.

The following is an example of a command definition

```
dir (<cardNum>|<srchOptions>) [displayOptions]
```

The `dir` command takes either a card number of a search specification, followed by display options.

Here are two examples of the `dir` command sent from the console window:

```
dir 0 -a
dir -t rsrc
```

Specifying Command Options

All command options and some command parameters are specified as flags that begin with a dash. For example:

```
-c
-enable
```

Some flags are followed by a keyword or value. You must leave white space between the flag and the value. For example:

```
-f D:\temp\myLogFile
-t Rsrc
```

NOTE: You use the dash (-) character to specify options for console commands. If you are using Palm Debugger, you must use the backslash (\) character to specify options for commands that you type in the debugging window; this is because the expression parser used for debugging commands interprets the dash as a minus sign.

Specifying Numeric and Address Values

Many of the console commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (#) or binary (%). Table 4.3 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 4.3 Specifying numeric values in Palm Debugger

Hex value	Decimal value	Binary value
`64 or $64`	`#100`	`%01100100`
`F5 or $F5`	`#245`	`%11110101`
`100 or $100`	`#256`	`%100000000`

Console Window Commands

You use the console window to send commands to the console nub that is running on the handheld device.

This section provides a description of all of the commands in alphabetical order. For convenience, the commands are categorized here:

Table 4.4 Console window command categories

Command category	Commands
Card Information	CardFormat, CardInfo, and StoreInfo.
Chunk Utility	Free, Info, Lock, New, Resize, SetOwner, and Unlock.
Database Utility	Close, Create, Del, Dir, Export, Import, Open, Opened, and SetInfo.
Debugging Utility	DM, GDB, MDebug, and SB.
Gremlin	Gremlin and GremlinOff.
Heap Utility	HC, HChk, HD, HF, HI, HL, HS, HT, and HTorture.
Host Control	Help, Log, and SaveImages.

Table 4.4 Console window command categories

Command category	Commands
Miscellaneous Utility	SimSync and SysAlarmDump.
Record Utility	AddRecord, , , DelRecord, DetachRecord, FindRecord, ListRecords, MoveRecord, and SetRecordInfo.
Resource Utility	AddResource, AttachResource, ChangeResource, DelResource, DetachResource, , ListResources,and SetResourceInfo.
System	Battery, ColdBoot, Doze, Exit, Feature, KInfo, Launch, Performance, PowerOn, Reset, Sleep, and Switch.

AddRecord

Purpose Adds a record to a database.

Usage `addrecord <accessPtr> <index> <recordText>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.
`recordText`	The record data.

AddResource

Purpose Adds a resource to a database.

Usage `addresource <accessPtr> -t <type> -id <id> <resourceText>`

Parameters

`accessPtr`	A pointer to the database.
`type`	The type of the resource that you are adding.
`id`	The ID for the resource that you are adding.
`resourceText`	The resource data.

AttachRecord

Purpose Attaches a record to a database.

Usage

```
attachrecord <accessPtr> <recordHandle> <index>
[options]
```

Parameters

`accessPtr`	A pointer to the database.
`recordHandle`	A handle to the record that you are attaching to the database.
`index`	The index of the record.
`options`	Optional. You can specify the following option: `-r` Replaces the existing record with the same index, if one exists.

AttachResource

Purpose Attaches a resource to a database.

Usage

```
attachrecord <accessPtr> <recordHandle> <index>
[options]
```

Parameters

`accessPtr`	A pointer to the database.
`recordHandle`	A handle to the resource that you are attaching to the database.
`index`	The index of the resource.
`options`	Optional. You can specify the following option: `-r` Replaces the existing resource with the same index, if one exists.

Battery

Purpose A battery utility command for performing battery operations.

Usage `battery [options]`

Parameters

`options`	Optional. Specifies the battery operation to perform. Use one of the following values:
	`-rStart <deltaSeconds>`
	Start radio charging in the number of seconds specified by `deltaSeconds`.
	`-rStop`
	Stop radio charging.
	`-rLoaded` (`yes` | `no`)
	Set loaded state to `yes` or `no`.

Example

```
battery -rStop
```

CardFormat

Purpose Formats a memory card.

Usage `cardformat <cardNum> <cardName> <manufName> <ramStoreName>`

Parameters

`cardNum`	The card number.
`cardName`	The name to associate with the card.
`manufName`	The manufacturer name to associate with the card.
`ramStoreName`	The RAM store name to associate with the card.

CardInfo

Purpose Displays information about a memory card.

Usage `cardinfo <cardNum>`

Parameters

`cardNum`	The card number about which you want information. You can use `0` to specify the built-in RAM.

Example

```
cardinfo 0

Name: PalmCard
Manuf: Palm Computing
Version: 0001
CreationDate: B1243780
ROM Size: 00118FFC
RAM Size: 00200000
Free Bytes : 0015ACB2
Number of heaps: #3
```

ChangeRecord

Purpose Replaces a record in a database.

Usage `changerecord <accessPtr> <index> <recordText>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.
`recordText`	The new record data.

ChangeResource

Purpose Replaces a resource in a database.

Usage `changeresource <accessPtr> <index> <recordText>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.
`resourceText`	The new resource data.

Close

Purpose Closes a database.

Usage `close <accessPtr>`

Parameters

`accessPtr`	A pointer to the database.

ColdBoot

Purpose Initiates a hard reset on the handheld device.

Usage `coldboot`

Parameters None

Comments Use the coldboot command to perform a hard reset of the handheld device. A hard reset erases all data on the device, restoring it to its new condition.

The handheld device requires confirmation of this operation. You are prompted to press the "UP" button on the device to confirm that you want to perform a hard reset, or press any other button to cancel the operation.

Example

```
coldboot
```

Create

Purpose Creates a new database on the handheld device.

Usage `create <cardNum> <name> [options]`

Parameters

`cardNum`	The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.
`name`	The name for the new database on the handheld device.
`options`	Optional. Specifies information about the new database: `-t <type>` The 4-character database type identifier. `-c <creator>` The 4-character database creator ID. `-v <version>` The database version number. `-r` Specify to indicate that the database is a resource database.

Comments Use the create command to create a new record or resource database on the handheld device.

Example

Del

Purpose Deletes a database from the handheld device.

Usage `del <cardNum> <fileName>`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.

`fileName`	The name of the database on the handheld device. Note that you must quote the database name if it contains spaces.

Comments Use the `del` command to delete a database from the specified card on the handheld device.

You can get a list of the databases on the device with the Dir command.

You cannot delete an open database.

Result If the database you want to delete is not found or is currently opened, you receive an error message.

Example

```
del 0 birds.pdb

Success!!
```

DelRecord

Purpose Deletes a record from a database.

Usage `delrecord <accessPtr> <index>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.

Comments Use the `delrecord` command to delete the record at the specified `index` value from the database specified by `accessPtr`.

DelResource

Purpose Deletes a resource from a database.

Usage `delresource <accessPtr> <index>`

Parameters

`accessPtr`	A pointer to the database.

`index`	The index of the resource in the database.

Comments Use the `delresource` command to delete the resource at the specified `index` value from the database specified by `accessPtr`.

DetachRecord

Purpose Detaches a record from a database.

Usage `detachrecord <accessPtr> <index>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.

Comments Use the `detachrecord` command to detach the record at the specified `index` value from the database specified by `accessPtr`.

DetachResource

Purpose Detaches a resource from a database.

Usage `detachresource <accessPtr> <index>`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.

Comments Use the `detachresource` command to detach the resource at the specified `index` value from the database specified by `accessPtr`.

Dir

Purpose Displays a list of the databases on the handheld device.

Usage

```
dir (<cardNum>|<searchOptions>) [<displayOptions>]
```

Parameters

`cardNum` The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.

`searchOptions` Optional. Options for listing a specific database. Specify any combination of the following flags.

- `-c <creatorID>`
  Search for a database by creator ID.
- `-latest`
  List only the latest version of each database.
- `-t <typeID>`
  Search for a database by its type.

`displayOptions` Optional. Options for which information is displayed in the listing. Specify any combination of the following flags.

`-a`	Show all information.
`-at`	Show the database attributes.
`-d`	Show the database creation, modification, and backup dates.
`-i`	Show the database appInfo and sortInfo field values.
`-id`	Show the database chunk ID
`-s`	Show the database size
`-m`	Show the database modification number.
`-n`	Show the database name.
`-r`	Show the number of records in the database.

`-tc`	Show the database type ID and creator ID.
`-v`	Show the database version number.

Comments

Use the dir command to display a list of the databases on a specific card or in the handheld device built-in RAM. You typically use the following command to list all of the databases stored in RAM on the handheld device:

```
dir 0
```

Or use the -a switch to display all of the information for each database:

```
dir 0 -a
```

Example

```
dir 0
```

```
name                    ID        total       data
---------------------------------------------------
*System             00D20A44  392.691 Kb  390.361
Kb
*AMX                00D209C4   20.275 Kb   20.123
Kb
*UIAppShell         00D20944    1.327 Kb    1.175
Kb
*PADHTAL Library    00D208E2    7.772 Kb    7.674
Kb
*IrDA Library       00D20876   39.518 Kb   39.402
Kb
 ...
 MailDB             0001817F    1.033 Kb    0.929
Kb
 NetworkDB          0001818B    0.986 Kb    0.722
Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842
Kb
 DatebookDB         000181FB    0.084 Kb    0.000
Kb
---------------------------------------------------
-
Total: 41
```

DM

Purpose Displays a range of memory values.

Usage `dm <addr> [<count>]`

Parameters

`addr`	The starting memory address to be displayed.
`count`	The number of bytes to be displayed. If this is omitted, eight bytes of data are displayed.

Example

```
dm 0000f000
```

```
0000F000: 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 ................
```

Doze

Purpose Instructs the handheld device's CPU to sleep while maintaining the peripherals and the clock.

Usage `doze [options]`

Parameters

`options`	You can optionally specify the following flags: `-light` The handheld device will awaken in response to any interrupt.

Example

```
doze -light
```

Exit

Purpose Exits the debugger.

Usage `exit`

Parameters None.

Export

Purpose Copies a Palm OS database from the handheld device to the desktop computer.

Usage `export <cardNum> <fileName>`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
`fileName`	The name of the database on the handheld device. Note that you must quote the database name if it contains spaces.

Comments Use the `export` command to copy a database from the handheld device to your desktop computer. You can get a list of the databases on the device with the Dir command.

If the database contains resources, it is copied in standard PRC format; if the database contains records, it is copied in standard PDB format. Note that these two formats are actually identical.

The exported file is stored in the `Device` subdirectory of the directory in which Palm Debugger executable is stored.

The exported file is named `fileName`, with no added extensions.

Example

```
export 0 "Text to Hex"

Exporting resource 'code'=0....
Exporting resource 'data'=0....
Exporting resource 'pref'=0....
Exporting resource 'rloc'=0....
Exporting resource 'code'=1....
Exporting resource 'tFRM'=1000....
Exporting resource 'tver'=1....
Exporting resource 'tAIB'=1000....
Exporting resource 'Tbmp'=1000....
Exporting resource 'Tbmp'=1001....
Exporting resource 'MBAR'=1000....
Exporting resource 'Talt'=1000....
```

```
Exporting resource 'Talt'=1001....
Success!!
```

Feature

Purpose Accesses features.

Usage `feature [options]`

Parameters

`options` Optional. You can use the following options:

`-all` Displays a list of all known features

`-unreg <creator> <num>`
Unregisters the specified feature

`-get <creator> <num>`
Displays the value of a feature

`-set <creator> <num> <value>`
Sets the value of a feature.

Example

```
feature -all

ROM:  creator   number   value
      'psys'     #1      03003000
      'psys'     #2      00010000
RAM:  creator   number   value
      'psys'     #3      00000001
      'psys'     #4      00000001
      'psys'     #7      00000001
      'netl'     #0      02003000
      'irda'     #0      03003000

feature -get psys 3

Value = 00000001
```

FindRecord

Purpose Finds a record by ID.

Usage `findrecord <accessPtr> <id>`

Parameters

`accessPtr`	A pointer to the database.
`id`	The unique record ID.

Free

Purpose Disposes of a chunk.

Usage `free (<hexChunkPtr> | localID>) [options]`

Parameters

`hexChunkPtr or localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`options` Optional. You can specify the following options:

`-card` <`cardNum`>
The card number if a local ID is specified instead of a chunk pointer.

GDB

Purpose Enables or disables Gdb debugging

Usage `gdb [options]`

Parameters

`options` Optional. You can specify the following options:

`-enable`
Enables gdb debugging.

`-disable`
Disables gdb debugging.

GetResource

Purpose Retrieves the specified resource.

Usage `getresource -t <type> -id <id>`

Parameters

`type`	The type of resource that you want to retrieve.
`id`	The ID of the resource that you want to retrieve.

Gremlin

Purpose Activates a gremlin until the specified event occurs.

Usage `gremlin <num> <until>`

Parameters

`num`	The number of the gremlin to activate.
`until`	The event that deactivates the gremlin.

GremlinOff

Purpose Deactivates the current gremlin.

Usage `gremlinoff`

Parameters None

Example

```
gremlinoff
```

HC

Purpose Compacts a memory heap.

Usage `hc <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap to be compacted. Heap number `0x0000` is always the dynamic heap.

Example

```
hc 0002
Heap Compacted
```

HChk

Purpose Checks the integrity of a heap.

Usage `hchk <heapId> [options]`

Parameters

`heapId`	The hexadecimal number of the heap whose contents are to be checked. Heap number `0x0000` is always the dynamic heap.
`options`	Optional. You can specify the following option:
	`-c` Check the contents of each chunk.

Example

```
hchk 0000
Heap OK
```

HD

Purpose Displays a hexadecimal dump of the specified heap.

Usage `hd <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap whose contents are to be displayed. Heap number `0x0000` is always the dynamic heap.

Comments Use the `hd` command to display a dump of the contents of a specific heap from the handheld device. You can use the HL command to display the heap IDs.

Example

```
  hd 0

Displaying Heap ID: 0000, mapped to 00001480
                     req   act
resType/  #resID/
 start    handle   localID   size   size  lck own
flags type  index attr ctg  uniqueID name
--------------------------------------------------
------------------------------------------
-00001534 00001494 F0001495 000456 00045E  #0  #0
fM Graffiti Private
-00001992 00001498 F0001499 000012 00001A  #0  #0
fM DataMgr Protect List (DmProtectEntryPtr*)
-000019AC 00001490 F0001491 00001E 000026  #0  #0
fM Alarm Table
-000019D2 0000148C F000148D 000038 000040  #0  #0
fM
*00001A12 0000149C F000149D 000396 00039E  #2  #1
fM Form "3:03 pm"
*00001DB0 000014A0 F00014A1 00049A 0004A2  #2  #0
fM
 00002252 -------- F0002252 00002E 00003E  #0  #0
FM
 00002290 -------- F0002290 00EC40 00EC50  #0  #0
FM
```

```
-00010EE0 -------- F0010EE0 000600 000608  #0 #15
fM Stack: Console Task

...

000114E8 -------- F00114E8 000FF8 001008  #0  #0
FM
-000124F0 -------- F00124F0 001000 001008  #0 #15
fM
-00017D30 -------- F0017D30 00003C 000044  #0 #15
fM SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15
fM Feature Manager Globals (FtrGlobalsType)
-00017D84 -------- F0017D84 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Update 3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15
fM DmOpenRef: 'Update 3.0.2'
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15
fM Handle Table: 'Ô©Update 3.0.2'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15
fM DmOpenInfoPtr: 'Ô©Update 3.0.2'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15
fM DmOpenRef: 'Ô©Update 3.0.2'
--------------------------------------------------
----------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14      (010C50 bytes)
  Movable Chunks:     #51      (005E80 bytes)
  Non-Movable Chunks: #0       (000000 bytes)
```

Help

Purpose Displays a list of commands or help for a specific command.

Usage

```
help
help <command>
```

Parameters

`command`	The name of the command for which you want help displayed.

Example

```
help hchk

Do a Heap Check.
Syntax: hchk <hex heapID> [options...]
         -c    : Check contents of each chunk
```

HF

Purpose Allocates almost all of the free bytes in a heap, reserving the specified amount of free space.

Usage

```
hf <heapId> <freeBytes>
```

Parameters

`heapId`	The hexadecimal number of the heap. Heap number `0x0000` is always the dynamic heap.
`freeBytes`	The number of bytes to leave unallocated.

Example

```
hf 0000 20
```

HI

Purpose Initializes the specified memory heap.

Usage `hi <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap to be initialized. Heap number `0x0000` is always the dynamic heap.

Example

```
hi 0006
```

HL

Purpose Displays a list of memory heaps.

Usage `hl <cardNum>`

Parameters

`cardNum`	The card number on which the heaps are located. You almost always use `0` to specify the built-in RAM.

Comments Use the `hl` command to list the memory heaps in built-in RAM or on a card.

Example

```
hl 0

   index   heapID  heapPtr     size     free
maxFree    flags
-----------------------------------------------------
----------
       0     0000 00001480 00016B80 00010C50
0000EC48     8000
       1     0001 1001810E 001E7EF2 0014AD6A
00147D3A     8000
       2     0002 10C08212 00118DEE 0000A01C
0000A014     8001
```

HS

Purpose Scrambles the specified heap.

Usage `hs <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap to be scrambled. Heap number `0x0000` is always the dynamic heap.

Comments Scrambling a heap moves its contents around. You can use this to verify that the program is using handles in the prescribed manner.

Example

```
hs 0002
heap scrambled
```

HT

Purpose Displays summary information for the specified heap.

Usage `ht <heapId>`

Parameters

`heapId`	The hexadecimal number of the heap to be scrambled. Heap number `0x0000` is always the dynamic heap.

Comments The `ht` command displays the summary information that is also shown at the end of a heap dump generated by the HD command.

Example

```
ht 0000
Displaying Heap ID: 0000, mapped to 00001480
--------------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010CAA bytes)
```

```
Movable Chunks:     #48     (005E26 bytes)
Non-Movable Chunks: #0      (000000 bytes)
```

HTorture

Purpose Tortures a heap to test its integrity.

Usage `htorture <heapId> [options]`

Parameters

`heapId` The hexadecimal number of the heap to be tortured. Heap number `0x0000` is always the dynamic heap.

`options` Optional. You can specify a combination of the following options:

- `-c` Checks the contents of every chunk.
- `-f <number>` Reports if the heap is filled beyond the specified percentage. The default is `90` percent.
- `-l <filename>` Specifies the name of the log file
- `-m <hexSize>` The maximum chunk size. The default value is `0x400`.
- `-p <level>` The progress level to display. Specify a number between `0` (minimum detail) and `2` (maximum detail). The default value is `0`.

Comments Use the `htorture` command to torture-test a memory heap. You can specify a logging file to which the output of the test is sent. You can also use the `-p` command to control how progress is displayed.

Import

Purpose Copies a Palm OS database from the desktop computer to the handheld device.

Usage `import <cardNum> <fileName>`

Parameters

`cardNum`	The card number on which the database is to be installed. You almost always use `0` to specify the built-in RAM.
`fileName`	The name of the file on the desktop computer. You can specify an absolute file name path, or a relative file name path. The default search path is the `Device` subdirectory of the directory in which Palm Debugger executable is stored.

Comments Use the `import` command to load a new version of your application or database onto the handheld device.

This command provides a more convenient install operation and has the same functionality as the installer tool provided with the HotSync Manager application.

The name of the database on the handheld device is the name stored in the file, and is not the same as the file name. If a database with a matching name is already open on the handheld device, an error is generated. If a database with a matching name is already stored on the handheld device, that database is deleted and replaced by the file.

Result If a database with a matching name is currently open on the handheld device, the `dmErrAlreadyExists` error code (`0x0219`) is generated.

Example

```
import 0 Tex2HexApp.prc

Creating Database on card 0
name: Text to Hex
```

```
type appl, creator TxHx

Importing resource 'code'=0....
Importing resource 'data'=0....
Importing resource 'pref'=0....
Importing resource 'rloc'=0....
Importing resource 'code'=1....
Importing resource 'tFRM'=1000....
Importing resource 'tver'=1....
Importing resource 'tAIB'=1000....
Importing resource 'Tbmp'=1000....
Importing resource 'Tbmp'=1001....
Importing resource 'MBAR'=1000....
Importing resource 'Talt'=1000....
Importing resource 'Talt'=1001....
Success!!
```

Info

Purpose Displays information about a memory chunk.

Usage `info (<hexChunkPtr> | localID>) [options]`

Parameters

`hexChunkPtr` or `localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`options`
Optional. You can specify the following options:

`-card` <`cardNum`>
The card number if a local ID is specified instead of a chunk pointer.

KInfo

Purpose Displays a list of all system kernel information.

Usage `kinfo [options]`

Parameters

`options` Optional. Specify the kernel information that you want to see displayed. Use a combination of the following flags:

`-all`
Display all kernel information.

`-task (<id> | all)`
Display task information.

`-sem (<id> | all)`
Display semaphore information.

`-tmr (<id> | all)`
Display timer information.

Comments Use the kinfo command to display a list of system kernel information, including tasks, semaphores, event groups, and timers.

Example

```
  kinfo -all

Task Information:
  taskID     tag  priority  stackPtr   status
  ------------------------------------------------
---------------------
  000176EA    AMX  #   0    00017556   Idle:
Waiting for Trigger
  000178BE   psys  #  30    00013364   Waiting on
event timer
  0001795A   CONS  #  10    0001103E   Running

Semaphore Information:
 semID     tag  type     initValue  curValue
nesting    ownerID
  ------------------------------------------------
---------------------------
```

```
  000177EE   MemM  resource   #-1        #1
(free)      #0       00000000
  00017822   SlkM  counting   #1         #1
(avail.)    #0       00000000
  0001788A   SndM  counting   #1         #1
(avail.)    #0       00000000
  00017A5E   SerM  counting   #0         #0
(unavail.)  #0       00000000

Timer Information:
  tmrID      tag   ticksLeft   period   procPtr
  ---------------------------------------------
  000177BA   psys    #  83    #   0    10C6C618
```

Launch

Purpose Launches an application on the handheld device.

Usage `launch [-t] [-ns] [-ng] <cardNum> <name> [<cmd> <cmdStr>`

Parameters

`-t`	Launches the application as a separate task.
`-ns`	Use the caller’s stack.
`-ng`	Use the caller’s globals environment.
`cardNum`	The card number on which application is located. You almost always use `0` to specify the built-in RAM.
`name`	The name of the application to be launched.
`cmd`	Optional. Use to specify a command for the application.
`cmdStr`	Optional. Use to specify an arguments string for `cmd`.

ListRecords

Purpose Lists the records in a database.

Usage `listrecords <accessPtr>`

Parameters

`accessPtr` A pointer to the database.

ListResources

Purpose Lists the resources in a database.

Usage `listresources <accessPtr>`

Parameters

`accessPtr` A pointer to the database.

Lock

Purpose Locks a memory chunk.

Usage `lock (<hexChunkPtr> | localID>) [options]`

Parameters

`hexChunkPtr or localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`options` Optional. You can specify the following options:

`-card` <`cardNum`>
The card number if a local ID is specified instead of a chunk pointer.

Log

Purpose Toggles logging of debugger output to a file.

Usage `log <fileName>`

Parameters

`fileName`	The name of the file to which debugger output is sent.

Comments Use the `log` command to start or stop logging of debugger output to a file.

MDebug

Purpose Sets the Memory Manager debug mode, which you can use to track down memory corruption problems.

Usage `mdebug [options]`

Parameters

`options` — Optional. Specify the kernel information that you want to see displayed. Use a combination of the following flags:

`-full`
: Shortcut for full debugging.

`-partial`
: Shortcut for partial debugging.

`-off`
: Shortcut to disable debugging.

`-a`
: Check / scramble all heaps each time.

`-a-`
: Check only the heap currently in use.

`-c`
: Check heap(s) on some memory calls.

`-ca`
: Check heap(s) on all memory calls.

`-c-`
Do not check heaps.

`-f`
Check free chunk contents.

`-f-`
Do not check free chunk contents.

`-min`
Store minimum available free space in dynamic heap in the global variable `GMemMinDynHeapFree`.

`-min-`
Do not record minimum free space.

`-s`
Scramble heap(s) on some memory calls.

`-sa`
Scramble heap(s) on all memory calls.

`-s-`
Do not scramble heaps.

Comments Use the `mdebug` command to enable debugging for tracking down memory corruption problems.

WARNING! The different debug modes enabled by mdebug can significantly slow down operations on the handheld device. Full checking is slowest, partial checking is slow, and only enabling specific options is the fastest.

Example

```
mdebug -full
Current mode = 003A
  Every heap checked/scrambled per call
  Heap(s) checked on EVERY Mem call
  Heap(s) scrambled on EVERY Mem call
  Free chunk contents filled & checked
  Minimum dynamic heap free space recording OFF
```

MoveRecord

Purpose Moves a record in the database by changing its index.

Usage `moverecord <accessPtr> <fromIndex> <toIndex>`

Parameters

`accessPtr`	A pointer to the database.
`fromIndex`	The original index of the record in the database.
`toIndex`	The new index for the record in the database.

New

Purpose Allocates a new chunk in a heap.

Usage `new <heapId> <hexChunkSize> [options]`

Parameters

`heapId` The hexadecimal number of the heap in which to allocate a new chunk. Heap number `0x0000` is always the dynamic heap. Note that heapId is ignored if you specify the `-near` option.

`hexChunkSize` The number of bytes in the new chunk, specified as a hexadecimal number.

`options` Optional. You can specify a combination of the following options:

- `-c` Fill the chunk contents.
- `-lock` Pre-lock the chunk.
- `-n` Make the chunk unmoveable.
- `-near <ptr>` Allocate the new chunk in the same heap as the specified pointer. If this option is specified, the `heapId` is ignored.
- `-o <ownerId>` Set the owner of the chunk to the specified ID value.

Open

Purpose Opens a database.

Usage `open <cardNum> <name> [options]`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
`name`	The name of the database.
`options`	Optional. You can specify the following options: `-r` Open the database for read-only access. `-p` Leave the database open.

Opened

Purpose Lists all of the currently opened databases.

Usage `opened`

Parameters None.

Example

```
  opened

name                  resDB   cardNum   accessP
ID    openCnt      mode
--------------------------------------------------
----------------------------
*Graffiti ShortCutsyes         0 00017D5C
001FFE7F          1       0007
*System          yes         0 00017FEE 00D20A44
1       0005
--------------------------------------------------
----------------------------
Total: 2 databases opened
```

Performance

Purpose Sets the performance level of the handheld device.

Usage `performance [options]`

Parameters `options` You can specify the following options:

`-b <baud>`
Uses the specified <baud> rate to calculate the nearest clock frequency value.

`-d <duty>`
Set the CPU duty cycle. The <duty> value specifies the number of CPU cycles out of every 31 system clock ticks.

`-f <freq>`
Set the system clock frequency to the specified Hz value; select the nearest baud multiple as the frequency.

`-ff <freq>`
Set the system clock frequency to the specified Hz value; do not pick the nearest baud multiple.

PowerOn

Purpose Powers on the handheld device.

Usage `poweron`

Parameters None.

Example

```
poweron
```

Reset

Purpose Performs a soft reset on the handheld device.

Usage `reset`

Parameters None.

Comments This command performs the same reset that is performed when you press the recessed reset button on a Palm Computing handheld device.

Example

```
reset
Resetting system
```

Resize

Purpose Resizes an existing memory chunk.

Usage `resize (<hexChunkPtr> | localID>) <hexNewSize> [options]`

Parameters

`hexChunkPtr` or `localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`hexNewSize` The new size of the chunk, in bytes.

`options` Optional. You can specify the following options:

- `-c` Checks and fills the contents of the resized chunk.
- `-card` <`cardNum`>
  The card number if a local ID is specified instead of a chunk pointer.

SaveImages

Purpose Saves a memory card image.

Usage `saveimages`

Parameters None.

SB

Purpose Sets the value of a byte in memory.

Usage `sb <addr> <value>`

Parameters

`addr`	The address of the byte.
`value`	The new value of the byte.

SetInfo

Purpose Sets new information values for a database.

Usage `setinfo <cardNum> <dbName> [options]`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
dbName	The name of the database.
options	Options. You can specify a combination of the following values: -m <modification> Sets the modification number for the database. -n <name> Sets the name of the database.

-v <version>
Sets the version number of the database.

SetOwner

Purpose Sets the owner ID of a memory chunk.

Usage
```
setowner (<hexChunkPtr> | <localID>) <owner>
[options]
```

Parameters

`hexChunkPtr` or `localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`hexNewSize` The new size of the chunk, in bytes.

`owner` The new owner ID for the chunk.

`options` Optional. You can specify the following options:

`-card` <cardNum>
The card number if a local ID is specified instead of a chunk pointer. Use `0` to specify the built-in RAM.

SetRecordInfo

Purpose Changes information for a record in a database.

Usage
```
setrecordinfo <accessPtr> <index> [options]
```

Parameters

`accessPtr` A pointer to the database.

`index` The index of the record in the database.

`options` Optional. You can specify a combination of the following options:

`-a <hexAttr>`
Sets attribute bit settings for the record.

`-u <uniqueId>`
Sets unique record ID for the record.

SetResourceInfo

Purpose Changes information for a resource in a database.

Usage `setresourceinfo <accessPtr> <index> [options]`

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.
`options`	Optional. You can specify a combination of the following options: `-t <resType>` Sets resource type for the resource. `-id <resId>` Sets resource ID for the resource.

SimSync

Purpose Simulates a synchronization operation on a specific database.

Usage `simsync <accessPtr>`

Parameters

`accessPtr`	A pointer to the database.

Sleep

Purpose Shuts down all peripherals, the CPU, and the system clock.

Usage `sleep`

Parameters None.

StoreInfo

Purpose Displays information about a memory store.

Usage `storeinfo <cardNum>`

Parameters

`cardNum`	The card number for which you want information displayed. You almost always use `0` to specify the built-in RAM.

Example

```
storeinfo 0

ROM Store:
  version: 0001
  flags: 0000
  name: ROM Store
  creation date: 00000000
  backup date: 00000000
  heap list offset: 00C08208
  init code offset1: 00C0D652
  init code offset2: 00C1471E
  database dirID: 00D20F7E

RAM Store:
  version: 0001
  flags: 0001
  name: RAM Store 0
  creation date: 00000000
  backup date: 00000000
  heap list offset: 00018100
  init code offset1: 00000000
  init code offset2: 00000000
  database dirID: 0001811F
```

Switch

Purpose Switches the application that is used to provide the user interface on the handheld device.

Usage `switch <cardNum> <name> [<cmd> <cmdStr>]`

Parameters

`cardNum`	The number of the card on which the user interface application is stored. You almost always use `0` to specify the built-in RAM.
`name`	The name of the application.
`cmd`	Optional. Use to specify a command for the application.
`cmdStr`	Optional. Use to specify an arguments string for `cmd`.

SysAlarmDump

Purpose Displays the system alarm table.

Usage `sysalarmdump`

Parameters None.

Example

```
  sysalarmdump

                              alarm
card
   date    time    ref     seconds    dbID    #
quiet  triged noted
--------------------------------------------------
------------------------
 7/29/1999  00:00  00000000  B3C54A00  00D1FCF8
4004  false  false  false
 1/ 1/1904  00:00  00000000  00000000  00000000
0000  false  false   true
```

Unlock

Purpose Unlocks a memory chunk.

Usage `unlock (<hexChunkPtr> | localID>) [options]`

Parameters

`hexChunkPtr` or `localID`
A pointer to a chunk in memory, or the ID of a chunk on the specified card number.

`options`
Optional. You can specify the following options:

`-card <cardNum>`
The card number if a local ID is specified instead of a chunk pointer.

Console Command Summary

Card Information Commands

`CardFormat`	Formats a memory card.
`CardInfo`	Retrieves information about a memory card.
`StoreInfo`	Retrieves information about a memory store.

Chunk Utility Commands

`Free`	Disposes of a heap chunk.
`Info`	Displays information on a heap chunk.
`Lock`	Locks a heap chunk.
`New`	Allocates a new chunk in a heap.
`Resize`	Resizes an existing heap chunk.
`SetOwner`	Sets the owner of a heap chunk.
`Unlock`	Unlocks a heap chunk.

Database Utility Commands

`Close`	Closes a database.
`Create`	Creates a new database.
`Del`	Deletes a database.
`Dir`	Lists the databases.
`Export`	Exports a database to the desktop computer.
`Import`	Imports a database from the desktop computer.
`Open`	Opens a database.
`Opened`	Lists all currently opened databases.
`SetInfo`	Sets database information, such as its name, version number, and modification number.

Debugging Utility Commands

`DM`	Displays memory.
`GDB`	Enables or disables Gdb debugging.
`MDebug`	Sets the Memory Manager debug mode.
`SB`	Sets the value of a byte.

Gremlin Commands

`Gremlin`	Activates the specified gremlin until a specified event occurs.
`GremlinOff`	Deactivates the current gremlin.

Heap Utility Commands

`HC`	Compacts a memory heap.
`HChk`	Checks a heap.
`HD`	Displays a dump of a memory heap.

`HF`	Allocates all free space in a memory heap, minus a specified number of bytes.
`HI`	Initializes a memory heap.
`HL`	Lists all of the memory heaps on the specified memory card.
`HS`	Scrambles a heap.
`HT`	Performs a heap total.
`HTorture`	Torture-tests a heap.

Host Control Commands

`Help`	Provides help on the console commands.
`Log`	Starts or stops logging to a file.
`SaveImages`	Saves an image of a memory card to file.

Miscellaneous Utility Commands

`SimSync`	Simulates a synchronization operation on a database.
`SysAlarmDump`	Displays the alarm table.

Record Utility Commands

`AddRecord`	Adds a record to a database.
`AttachRecord`	Attaches a record to a database.
`ChangeRecord`	Replaces a record in a database.
`DelRecord`	Deletes a record from a database.
`DetachRecord`	Detaches a record from a database.
`FindRecord`	Finds a record by its unique ID.
`ListRecords`	Lists all of the records in a database.

`MoveRecord`	Changes the index of a record.
`SetRecordInfo`	Sets record information, such as its ID and attributes.

Resource Utility Commands

`AddResource`	Adds a resource to a database.
`AttachResource`	Attaches a resource to a database.
`ChangeResource`	Replaces a resource in a database.
`DelResource`	Deletes a resource from a database.
`DetachResource`	Detaches a resource from a database.
`GetResource`	Retrieves a resource from a database.
`ListResources`	Lists all resources in a database.
`SetResourceInfo`	Sets resource information, such as its ID and resource type.

System Commands

`Battery`	Battery utility command for starting or stopping radio charging, and for setting the loaded status.
`ColdBoot`	Boots the handheld device.
`Doze`	Puts the CPU to sleep while keeping the peripherals and clock running on the handheld device.
`Exit`	Exits the console.
`Feature`	Displays, retrieves, registers, or unregisters features.
`KInfo`	Displays kernel information.
`Launch`	Launches an application.

`Performance`	Sets performance levels, such as the system clock frequency and CPU duty cycle.
`PowerOn`	Powers on the handheld device.
`Reset`	Resets the memory system and formats both cards.
`Sleep`	Shuts down all peripherals, the CPU, and the system clock.
`Switch`	Switches the current user interface application.

5

Using the Palm Simulator

This chapter describes the Palm Simulator application, which you can use to test and debug Palm OS® applications.

WARNING! The Simulator can only be run on Macintosh computers; there is no Simulator available for developers using Windows-based computers. If you are using Windows, you can use the Palm OS Emulator, which can run on Windows, Unix, and Macintosh computers. The Emulator is described in Chapter 1, "Using the Palm OS® Emulator."

About the Simulator

The Simulator simulates a Palm handheld device on a Macintosh computer, and allows you to test and debug your Palm OS application within the simulated environment. The Simulator provides a graphical representation of a Palm handheld device on the Macintosh screen, and supports user interactions that mimic actual stylus actions on a handheld device.

To use the Simulator, you need to build your Palm OS application with the Simulator as your target, instead of the hardware device, as described in Building a Project for Use With the Simulator.

Figure 5.1 shows the Simulator screen.

Figure 5.1 The Simulator screen

Using the Simulator is very much like using an actual device, with some differences, which are described in Differences Between the Simulator and Actual Hardware. Use the Simulator to test your application as follows:

- Click the mouse on the representation of the device's physical controls (including the silk-screened icons) to activate those controls as you would tap on controls on the actual handheld device.
- Click any of the menus, buttons, or other user interface items your application provides, just as you would tap on those items on the actual handheld device.
- Use the mouse to write Graffiti® in the Graffiti area of the simulated screen. Or you can enter characters by typing on your Macintosh keyboard.
- Use the function keys F9 - F12 to simulate the four buttons if you havve made special button assignments in your application. Otherwise, these buttons are not functional.

As you interact with the simulated interface, you can trace events, run Gremlins, and use the Console window to debug your application. For more information, see Using the Simulator.

The Simulator Compared to The Emulator

The Simulator presents a similar interface to the Palm OS Emulator, which is described in Chapter 1, "Using the Palm OS® Emulator." The Simulator runs your program faster than the Emulator, but the Simulator has a few limitations of which you need to be aware:

- The Simulator runs only on Macintosh computers.
- The Simulator allows your application to make calls and perform actions that do not work on Palm handheld devices, while the Emulator does not allow these calls or actions. For more information, see the next section, Differences Between the Simulator and Actual Hardware.

The Simulator does provide certain debugging advantages relative to the Palm OS Emulator, as follows:

- The time to compile and then test your application is reduced when using the Simulator: you simply build your application for the Simulator, and double-click on the application to test it.
- Gremlins run somewhat faster on the Simulator than they do with the Palm OS Emulator.
- Debugging is more robust with the Simulator than with the Palm OS Emulator.
- Running code conditionally is easier in the Simulator than in the Palm OS Emulator.

In summary, if you are building your application on a Macintosh computer, the Simulator tends to provide a faster environment for the initial debugging and testing of your application. Palm Computing recommends, however, that you run your application with the Palm OS Emulator before downloading it to an actual handheld device. This will help you to discover any calls your application is making that do not work with the Palm OS.

Differences Between the Simulator and Actual Hardware

The few differences between an application running under the Palm Simulator and one running on a Palm OS device can cause difficulties during debugging. In particular, the Simulator allows an application to do a few things that won't work on the device.

If your application runs under the Simulator but doesn't run on the handheld device, check for the potential problems shown in Table 5.1.

Table 5.1 Application problems due to Simulator and Palm OS differences

Cause of problem	Explanation
Application calls standard C run-time library functions	These calls work under the Simulator, but may not work on handheld devices. Memory management functions such as `malloc` and `free`, string operations such as `strcpy` and `strcmp`, mathematical functions such as `rand` and `cos`, and other library functions do not work with the Palm OS. Note that even if a standard C run-time library function does work with the Palm OS, it can unnecessarily enlarge your application.
Application writes to storage RAM without using the `DMWrite` function	The Palm OS enforces write protection, while the Simulator does not. Your application will work properly with the Simulator, but will generate a bus error when you run it on the handheld device.
Application accesses 16-bit or 32-bit memory values at odd addresses	The Simulator allows these memory accesses, which generate bus errors on the handheld device. You often encounter this error when working with packed data.
Application attempts a code jump of more than 32K bytes	The Palm OS does not allow jumps that exceed 32K bytes. You need to rearrange your code to avoid such jumps.
Application writes records that are larger than 64K bytes	The Simulator allows records that are larger than 64K bytes long, but the Palm OS does not allow these records, which prevent HotSync from working properly.

Table 5.1 Application problems due to Simulator and Palm OS differences *(continued)*

Cause of problem	Explanation
Application overflows the stack	Palm OS handheld devices provide a stack that is only 2K bytes long, while the Simulator runs in an environment that allows for a much larger stack. This problem commonly arises when your application uses a large amount of local data. You can work around this limitation by storing your data in global variables or allocated database chunks instead of using local storage.
Application pointer errors	Pointer errors tend to have a more dramatic effect on Palm handheld devices than on the Simulator because of the greater density of data in memory on the device. Bad pointer values, array overwrites, and related problems are more likely to destroy important data on the Palm handheld device than on the Macintosh computer.
Launch code problems	When the handheld device is reset, it sends certain launch codes to applications. The main body of your PilotMain function should be enveloped in the following conditional code: `if (cmd == launchcode)`

In addition, the user interface to the Simulator differs from the user interface on Palm handheld devices, as shown in Table 5.2.

Table 5.2 User interface differences

Difference	Description
Multiple applications unavailable with Simulator	The user can switch which application is running on a Palm handheld device by selecting a new application from the Applications menu, or by pressing one of the hardware buttons at the bottom of the device. The Simulator runs only the application that you have built for the Simulator target. .
Application buttons	The simulated buttons at the bottom of the Simulator display are only available if your application has made special assignments to them. These four buttons are simulated by the F9, F10, F11, and F12 keys on the Macintosh keyboard.
Scrolling buttons	The scroll up and scroll down buttons are simulated by the page up and page down keys on the Macintosh keyboard.

Simulator Menu Commands Summary

This section describes the Simulator menus in the same order as they appear on the Macintosh menu bar:

- File Menu
- Edit Menu
- Window Menu
- Replay Menu
- Gremlin Menu
- Serial Port Menu
- Panel Menu

File Menu

Table 5.3 describes the commands available on the File menu.

Table 5.3 File menu commands

Command	Description
Save Card 0...	Writes the contents of memory card 0 to a file. This command uses the standard Save dialog box to prompt you for the file name and location. The default name is `PilotCard 0`, and the default location is the current application folder. Card 0 is the Palm built-in RAM, on which all applications and add-on applications are stored.
Save Card 1...	Writes the contents of memory card 1 to a file.
Save Before Quitting	Saves a snapshot of the contents of both memory cards after `StopApplication` has been called. When an application exits, it saves information such as its preferences to the memory card. If you use this command, Simulator saves the files `Pilot Card 0` and `Pilot Card 1`.
Quit	Quits the application.

Edit Menu

The Edit menu offers the standard Mac OS editing commands for the Console window and the Event Trace window.

Window Menu

The Window menu provides access to two special windows: the Console window and the Event Trace window. You can close either of these windows by clicking the button in the top-left corner or by deselecting the window in the menu. Table 5.4 describes the Window menu commands.

Table 5.4 Window menu commands

Command	Description
Console	Activates the Console window, which is described in Chapter 4, "Using the Console Window."
Event Trace	Displays the Event Trace window. The Event Trace window displays the last 100 events generated by the system software and application. For more information, see Tracing Events.

Replay Menu

The Replay menu allows you to record pen and key events to a script file. You can then use the script file to replay the same events. This is useful for testing and repeating problem cases. For more information about using the Replay menu commands, see Scripting Pen and Key Events.

Table 5.5 describes the commands available on the Replay menu.

Table 5.5 Replay menu commands

Command	Description
Record	Begins recording pen and key events to a file. To stop recording, deselect this command.
Break	Inserts a stop into the script so it stops during replay. Does not stop the recording process.
Save As	Saves the recorded script to file. By default, the Simulator saves a script to a file with the extension `.LOG` whenever you stop recording. Use the **Save As** command to create an additional copy of that script file. The default file name is "Pilot Script."
Playback	Plays back a previously recorded script that you select with the standard Open dialog box.
Pause	Pauses playback of a script. This command is available during playback, but not during the recording process.

Table 5.5 Replay menu commands *(continued)*

Command	Description
Step	Plays back the next pen or key event, then pauses. This command is available during playback, but not during the recording process.
Realtime	Tries to execute the script at the rate at which it was recorded. With this option off, scripts execute as fast as possible. Realtime is useful when you need to test user interface elements that are timing dependent, such as repeating buttons.

Gremlin Menu

Gremlins are a facility to generate pseudo-random pen and key events. You can use Gremlins to reveal program problems. Each Gremlin is a unique sequence of random taps, strokes, and so on. Red lines indicate how the pen was moved over the screen by the Gremlin.

Although you can define a large number of gremlins, the Gremlin menu of the Simulator only allows you to run gremlin number 0. To run other gremlins, you need to activate the Console window and use its Gremlin command, as described in Gremlin in Chapter 4, "Using the Console Window."

Table 5.6 describes the commands available on the Gremlins menu.

Table 5.6 Gremlins menu commands

Command	Description
New	Runs Gremlin number 0. Iterates through all events in that Gremlin, running continuously.
Step	Performs the next Gremlin event, then stops.
Resume	Resumes running continuously after a step or stop.
Stop	Stops generating Gremlin events.

Serial Port Menu

The Serial Port menu allows you to select a Mac OS port to use when your application connects to another application with the serial port. Table 5.7 shows the choices available on the Serial Port menu.

Table 5.7 Serial port menu commands

Command	Description
Modem Port	Selects the Mac OS Modem port.
Printer Port	Selects the Mac OS Printer port.

Panel Menu

The Panel menu allows you to set modem and network preferences so that you can test applications that use TCP/IP. Table 5.8 shows the commands available on the Panel menu.

This menu is disabled unless your application's directory contains the files `Modem.prc` and `Network.prc`, which you can copy from the `PalmOS Libraries` folder. The Panel menu is also disabled if one of the panels is already running.

Table 5.8 Panel menu commands

Command	Description
Modem Panel	Displays the Modem panel from the Preferences application. Use this panel to enter the settings for the modem connected to your Macintosh computer. The Modem selection list only displays modems compatible with the Palm device, so it may not have a selection for your computer's modem. If this is the case, choose the Standard selection. You may need to change the initialization string.
Network Panel	Displays the Network panel from the Preferences application. Use this panel to enter network settings and to connect to the network.

Using the Simulator

This section describes how to perform various tasks with the Simulator, including:

- building a project for use with the Simulator

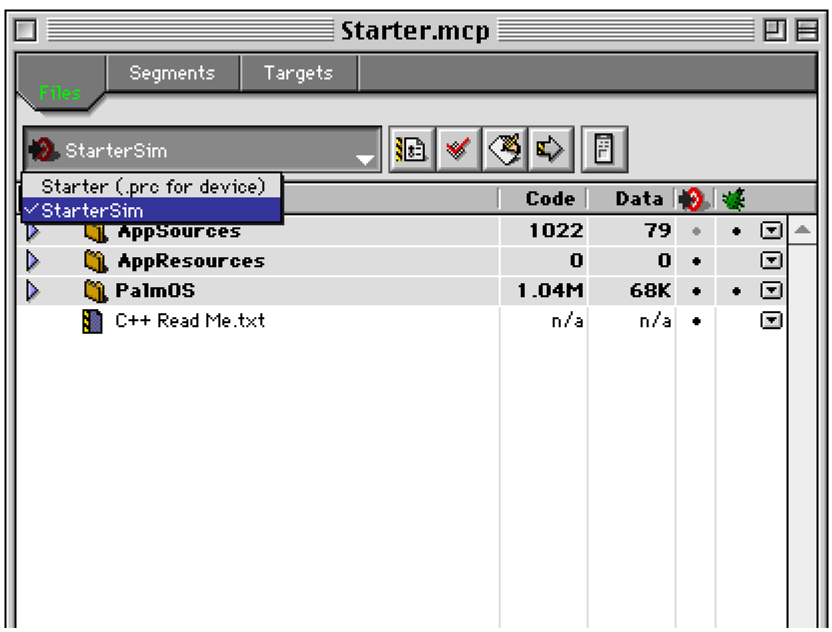
- tracing events
- scripting events for replay
- using gremlins
- testing communications
- saving memory images to file

Building a Project for Use With the Simulator

To use the Simulator, you need to build your Palm OS application in CodeWarrior with the Simulator libraries as your target, rather than targetting the Palm OS. When you activate the resulting executable, the Simulator starts up with your application loaded.

Figure 5.2 shows selecting the simulator target for the Starter project.

Figure 5.2 Choosing the simulator as a target



Most of the example projects provided with the Palm OS SDK have two targets: one that builds a Palm OS executable with a `.prc` suffix, and one that builds an executable with a `.mac` suffix that runs with the Simulator. When you select your target in

CodeWarrior, the Simulator target version ends with `Sim`. Table 5.9 shows several examples.

Table 5.9 Example project target names

Project File	Target name	Executable name
`Datebook.mcp`	`Datebook`	`Datebook.prc`
	`DatebookSim`	`Datebook.mac`
`Address.mcp`	`Address`	`Address.prc`
	`AddressSim`	`Address.mac`
`Starter.mcp`	`Starter`	`Starter.prc`
	`StarterSim`	`Starter.mac`

Using the Simulator With the CodeWarrior Debugger

If you want to set breakpoints or single-step through your applications' code, you can run the Palm Simulator from within the CodeWarrior Debugger. To do so, follow these steps:

1. Select the Enable Debugging command in the Project menu of the CodeWarrior IDE.
2. Build your project with the Simulator target. The resulting executable has the `.mac.SYM` extension.
3. Double-click the executable to start the CodeWarrior Debugger.

Tracing Events

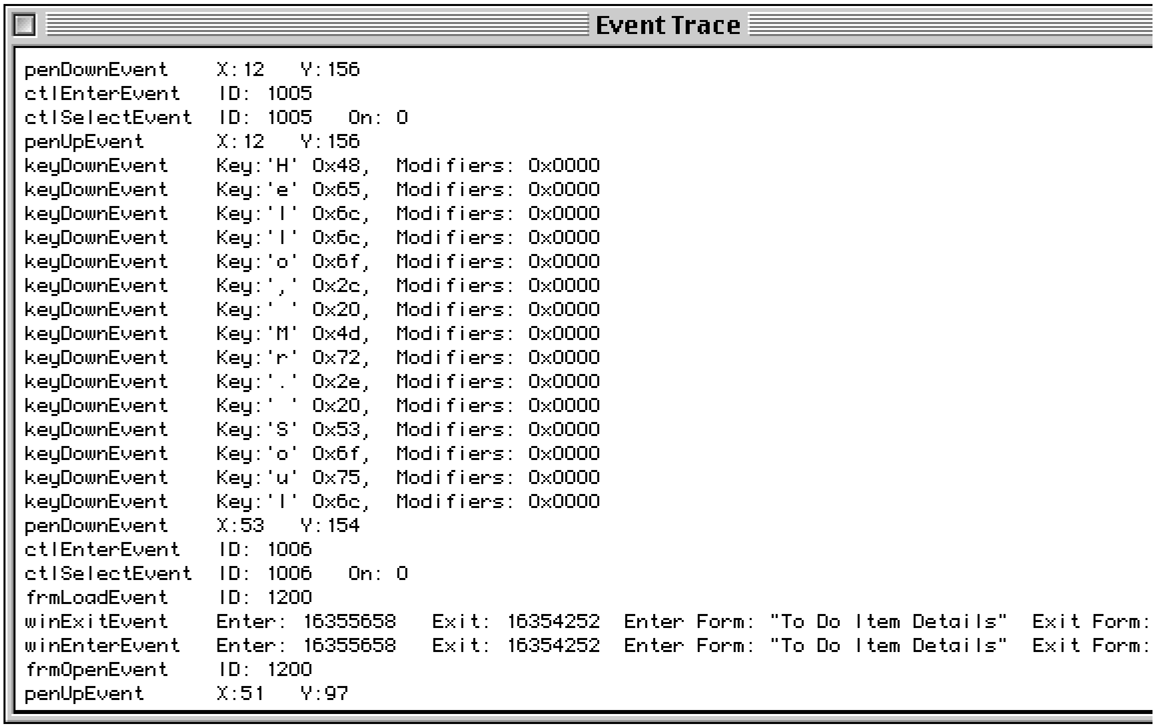
To trace the events that your application generates, select the **Event Trace** command from the Window menu. This displays the event tracing window until you once again select the **Event Trace** command to close the window.

The event trace shown in Figure 5.3 is the result of the following sequence of activities while running the Simulator version of the To Do list application:

1. Click the New button to create a new To Do item.
2. Type `Hello, Mr. Soul` on the keyboard

3. Click the Details... button to view the item details.

Figure 5.3 The event trace window



You can use the event tracing facility of the Simulator to verify that your application is properly receiving and processing key and pen events.

Scripting Pen and Key Events

The Simulator allows you to record user input events and save them in a script file for subsequent replay. You can replay the events in rapid order, or in realtime speed, which allows you to watch the processing of each event.

While recording the script, you can insert breaks, each of which causes the script to stop during replay.

To record a script, follow these steps:

1. Select the **Replay > Record** command. This begins the event recording

2. Record pen and key events.
3. Deselect the **Replay > Record** command.
4. Select the **Replay>Save As** command to save the recorded script to a file.

To playback a recorded script, follow these steps:

1. Select the **Replay > Playback** command. This displays the Open File dialog, which allows you to select a script to replay. The most recently saved script is always displayed as the default selection.
2. Select the **Replay>Realtime** command to play the script at the same speed at which it was recorded, or deselect the **Replay>Realtime** command to play the script rapidly.
3. Choose **Replay >Pause** and **Replay >Step** during replay to look in detail at the events that are executed.

Using Gremlins

The Simulator allows you to run a single gremlin, which is the gremlin numbered 0. To define and run other gremlins, you need to activate the Console window, and then run gremlins from that window. Gremlins are described in more detail in both Chapter 4, “Using the Console Window.” and Chapter 2, “Using Palm Debugger.”

To run gremlin 0 in the Simulator, follow these steps:

1. Select the **Gremlin>New** command to run gremlin 0. This command interates through all of the events in the gremlin and runs continously until stopped.
2. Select the **Gremlin>Step** command to perform the next gremlin event and then pause.
3. Select the **Gremlin>Resume** command to resume continuous execution of the gremlin events.
4. Select the **Gremlin>Stop** command to stop generating gremlin events.

Saving Memory Information to File

You can save the contents of the simulated built-in RAM by selecting the **File>Save Card 0...** command, which displays the standard Save dialog box and then saves the contents of RAM to a file. If the simulated device also has an extra memory card, you can use the **File>Save Card 1...** command to save the contents of that card to file.

If you select the **File>Save Before Quitting** command, the Simulator will automatically save the contents of the built-in RAM and of memory card 1 (if simulated) to file. The files are named `Pilot Card 0` and `Pilot Card 1`, respectively.

A

Debugger Protocol Reference

This appendix describes the debugger protocol, which provides an interface between a debugging target and a debugging host. For example, the Palm Debugger and the Palm OS® Emulator use this protocol to exchange commands and information.

IMPORTANT: This chapter describes the version of the Palm Debugger protocol that shipped on the Metrowerks CodeWarrior for the Palm Operating System, Version 6 CD-ROM. If you are using a different version, the features in your version might be different than the features described here.

About the Palm Debugger Protocol

The Palm debugger protocol allows a *debugging target*, which is usually a handheld device ROM or an emulator program such as the Palm OS Emulator, to exchange information with a *debugging host*, such as the Palm Debugger or the Metrowerks debugger.

The debugger protocol involves sending packets between the host and the target. When the user of the host debugging program enters a command, the host converts that command into one or more command packets and sends each packet to the debugging target. In most cases, the target subsequently responds by sending a packet back to the host.

Packets

There are three packet types used in the debugger protocol:

- The debugging host sends *command request packets* to the debugging target.

- The debugging target sends *command response packets* back to the host.
- Either the host or the target can send a *message packet* to the other.

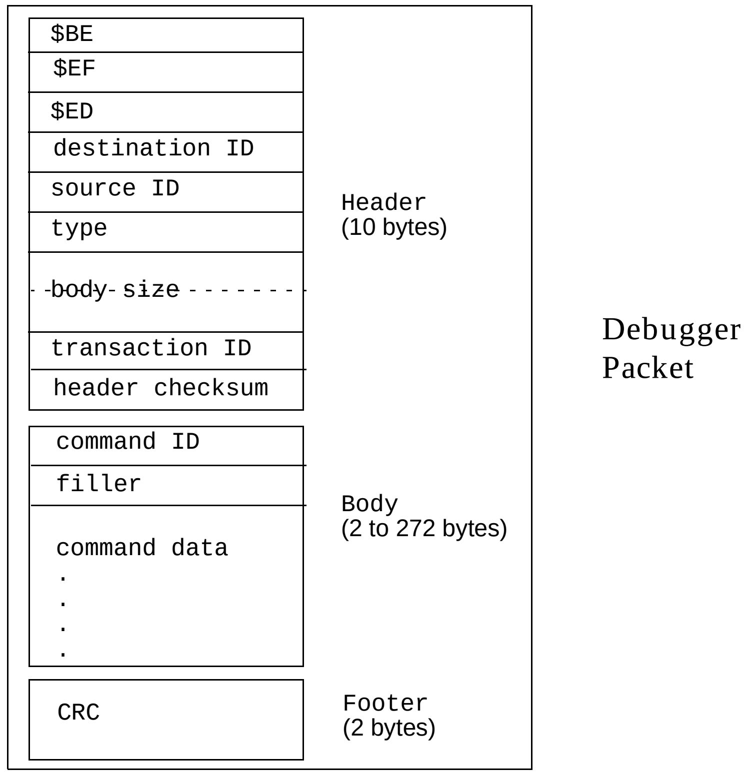
Although the typical flow of packets involves the host sending a request and the target sending back a response, although there are a some exceptions, as follows:

- The host can send some requests to the target that do not result in a response packet being returned. For example, when the host sends the `Continue` command packet to tell the target to continue execution, the target does not send back a response packet.
- The target can send response packets to the host without receiving a request packet. For example, whenever the debugging target encounters an exception, it sends a `State` response packet to the host.

Packet Structure

Each packet consists of a packet header, a variable-length packet body, and a packet footer, as shown in Figure A.1.

Figure A.1 Packet Structure



The Packet Header

The packet header starts with the 24-bit key value `$BEEFFD` and includes header information and a checksum of the header itself.

The Packet Body

The packet body contains the command byte, a filler byte, and between `0` and `270` bytes of data. See _SysPktBodyCommon for a description of the structure used to represent the two byte body header (the command and filler bytes), and see Table A.1 for a list of the command constants.

The Packet Footer

The packet footer contains a 16-bit CRC of the header and body. Note that the CRC computation does not include the footer.

Packet Communications

The communications protocol between the host and target is very simple: the host sends a request packet to the target and waits for a time-out or for a response from the target.

If a response is not detected within the time-out period, the host does not retry the request. When a response does not come back before timing out, it usually indicates that one of two things is happening:

- the debugging target is busy executing code and has not encountered an exception
- the state of the debugging target has degenerated so badly that it cannot respond

The host has the option of displaying a message to the user to inform him or her that the debugging target is not responding.

Constants

This section describes the constants and structure types that are used with the packets for various commands.

Packet Constants

```
#define sysPktMaxMemChunk256
#define sysPktMaxBodySize(sysPktMaxMemChunk+16)
#define sysPktMaxNameLen    32
```

`sysPktMaxMemChunk`
: The maximum number of bytes that can be read by the `Read Memory` command or written by the `Write Memory` command.

`sysPktMaxBodySize`
: The maximum number of bytes in a request or response packet.

`sysPktMaxNameLen`
: The maximum length of a function name.

State Constants

```
#define sysPktStateRspInstWords 15
```

`sysPktStateRespInstWords`
The number of remote code words sent in the response packet for the `State` command.

Breakpoint Constants

```
#define dbgNormalBreakpoints5
#define dbgTempBPIndex      dbNormalBreakpoints
#define dbgTotalBreakpoints (dbgTempBPIndex+1)
```

`dbgNormalBreakpoints`
The number of normal breakpoints available in the debugging target.

`dbgTempBPIndex`
The index in the breakpoints array of the temporary breakpoint.

`dbgTotalBreakpoints`
The total number of breakpoints in the breakpoints array, including the normal breakpoints and the temporary breakpoint.

Command Constants

Each command is represented by a single byte constant. The upper bit of each request command is clear, and the upper bit of each response command is set. Table A.1 shows the command constants.

Table A.1 Debugger protocol command constants

Command	Request constant	Response constant
Continue	sysPktContinueCmd	N/A
Find	sysPktFindCmd	sysPktFindRsp
Get Breakpoints	sysPktGetBreakpointsCmd	sysPktGetBreakpointsRsp
Get Routine Name	sysPktGetRtnNameCmd	sysPktGetRtnNameRsp

Table A.1 Debugger protocol command constants *(continued)*

Command	Request constant	Response constant
Get Trap Breaks	sysPktGetTrapBreaksCmd	sysPktGetTrapBreaksRsp
Get Trap Conditionals	sysPktGetTrapConditionalsCmd	sysPktGetTrapConditionalsRsp
Message	sysPktRemoteMsgCmd	N/A
Read Memory	sysPktReadMemCmd	sysPktReadMemRsp
Read Registers	sysPktReadRegsCmd	sysPktReadRegsRsp
RPC	sysPktRPCCmd	sysPktRPCRsp
Set Breakpoints	sysPktSetBreakpointsCmd	sysPktSetBreakpointsRsp
Set Trap Breaks	sysPktSetTrapBreaksCmd	sysPktSetTrapBreaksRsp
Set Trap Conditionals	sysPktSetTrapConditionalsCmd	sysPktSetTrapConditionalsRsp
State	sysPktStateCmd	sysPktStateRsp
Toggle Debugger Breaks	sysPktDbgBreakToggleCmd	sysPktDbgBreakToggleRsp
Write Memory	sysPktWriteMemCmd	sysPktWriteMemRsp
Write Registers	sysPktWriteRegsCmd	sysPktWriteRegsRsp

Data Structures

This section describes the data structures used with the request and response packets for the debugger protocol commands.

_SysPktBodyCommon

The `_SysPktBodyCommon` macro defines the fields common to every request and response packet.

```
#define _sysPktBodyCommon \
    Byte command; \
    Byte _filler;
```

Fields

`command` The 1-byte command value for the packet.

`_filler` Included for alignment only. Not used.

SysPktBodyType

The `SysPktBodyType` represents a command packet that is sent to or received from the debugging target.

```
typedef struct SysPktBodyType
{
    _SysPktBodyCommon;
    Byte data[sysPktMaxBodySize-2];
} SysPktBodyType;
```

Fields

`_SysPktBodyCommon`
The command header for the packet.

`data` The packet data.

SysPktRPCParamType

The SysPktRPCParamType is used to send a parameter in a remote procedure call. See the RPC command for more information.

```
typedef struct SysPktRPCParamInfo
{
    Byte  byRef;
    Byte  size;
    Word  data[?];
} SysPktRPCParamType;
```

Fields

`byRef` Set to `1` if the parameter is passed by reference.

`size` The number of bytes in the data array. This must be an even number.

`data` The parameter data.

BreakpointType

The `BreakpointType` structure is used to represent the status of a single breakpoint on the debugging target.

```
typedef struct BreakpointType
{
    Ptr     addr;
    Boolean enabled;
    Boolean installed;
} BreakpointType;
```

Fields

`addr`	The address of the breakpoint. If this is set to `0`, the breakpoint is not in use.
`enabled`	A Boolean value. This is `TRUE` if the breakpoint is currently enabled, and `FALSE` if not.
`installed`	Included for correct alignment only. Not used.

The Debugger Protocol Commands

This section describes each command that you can send to the debugging target, including a description of the response packet that the target sends back.

Continue

Purpose Tells the debugging target to continue execution.

Comments This command usually gets sent when the user specifies the `Go` command. Once the debugging target continues execution, the debugger is not reentered until a breakpoint or other exception is encountered.

NOTE: The debugging target does not send a response to this command.

Commands

The `Continue` request command is defined as follows:

```
#define sysPktContinueCmd0x07
```

Request Packet

```
typedef struct SysPktContinueCmdType
{
    _sysPktBodyCommon;
    M68KresgType  regs;
    Boolean       stepSpy;
    DWord         ssAddr;
    DWord         ssCount;
    DWord         ssCheckSum;
}SysPktContinueCmdType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `regs`
The new values for the debugging target processor registers. The new register values are stored in sequential order: `D0` to `D7`, followed by `A0` to `A6`.

--> `stepSpy`
A Boolean value. If this is `TRUE`, the debugging target continues execution until the value that starts at the specified step-spy address changes. If this is `FALSE`, the debugging target continue execution until a breakpoint or other exception is encountered.

--> `ssAddr`
The step-spy starting address. An exception is generated when the value starting at this address, for `ssCount` bytes, changes on the debugging target.

--> `ssCount`
The number of bytes in the “spy” value.

--> `ssCheckSum`
A checksum for the “spy” value.

Find

Purpose

Searches for data in memory on the debugging target.

Comments

.

Commands

The `Find` request and response commands are defined as follows:

```
#define sysPktFindCmd0x13
#define sysPktFindRsp0x93
```

Request Packet

```
typedef struct SysPktFindCmdType
{
    _sysPktBodyCommon;
    DWord     firstAddr;
    DWord     lastAddr;
    Word      numBytes
    Boolean   caseInsensitive;
    Byte      searchData[?];
}SysPktFindCmdType;
```

Fields

--> `_sysPktBodyCommon`
: The common packet header, as described in _SysPktBodyCommon.

--> `firstAddr`
: The starting address of the memory range on the debugging target to search for the data.

--> `lastAddr`
: The ending address of the memory range on the debugging target to search for the data.

--> `numBytes`
: The number of bytes of data in the search string.

--> `searchData`
: The search string. The length of this array is defined by the value of the `numBytes` field.

Response Packet

```
typedef struct SysPktFindRspType
{
    _sysPktBodyCommon;
```

```
    DWord   addr;
    Boolean found;
}SysPktFindRspType
```

Fields

<-- `_sysPktBodyCommon`
: The common packet header, as described in [_SysPktBodyCommon](#).

<-- `addr`
: The address of the data string in memory on the debugging target.

<-- `found`
: A Boolean value. If this is `TRUE`, the search string was found on the debugging target, and the value of addr is valid. If this is `FALSE`, the search string was not found, and the value of addr is not valid.

Get Breakpoints

Purpose Retrieves the current breakpoint settings from the debugging target.

Comments The body of the response packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible breakpoint.

If a breakpoint is currently disabled on the debugging target, the enabled field for that breakpoint is set to `0`.

If a breakpoint address is set to 0, the breakpoint is not currently in use.

The `dbgTotalBreakpoints` constant is described in [Breakpoint Constants](#).

Commands The `Get Breakpoints` command request and response commands are defined as follows:

```
#define sysPktGetBreakpointsCmd0x0B
#define sysPktGetBreakpointsRsp0x8B
```

Request Packet

```
typedef struct SysPktGetBreakpointsCmdType
{
  _sysPktBodyCommon;
}SysPktGetBreakpointsCmdType
```

Fields

--> _sysPktBodyCommon
The common packet header, as described in _SysPktBodyCommon.

Response Packet

```
typedef struct SysPktGetBreakpointsRspType
{
  _sysPktBodyCommon;
  BreakpointType db[dbgTotalBreakpoints];
}SysPktGetBreakpointsRspType
```

Fields

<-- _sysPktBodyCommon
The common packet header, as described in _SysPktBodyCommon.

<-- bp
An array with an entry for each of the possible breakpoints. Each entry is of the type BreakpointType.

Get Routine Name

Purpose

Determines the name, starting address, and ending address of the function that contains the specified address.

Comments

The name of each function is imbedded into the code when it gets compiled. The debugging target can scan forward and backward in the code to determine the start and end addresses for each function.

Commands

The `Get Routine Name` command request and response commands are defined as follows:

```
#define sysPktGetRtnNameCmd0x04
#define sysPktGetRtnNameRsp0x84
```

Request Packet

```
typedef struct SysPktRtnNameCmdType
{
    _sysPktBodyCommon;
    void*   address
}SysPktRtnNameCmdType;
```

Fields

--> `_sysPktBodyCommon`
: The common packet header, as described in _SysPktBodyCommon.

--> `address`
: The code address whose function name you want to discover.

Response Packet

```
typedef struct SysPktRtnNameRspType
{
    _sysPktBodyCommon;
    void*   address;
    void*   startAddr;
    void*   endAddr;
    char    name[sysPktMaxNameLen];
}SysPktRtnNameRspType;
```

Fields

<-- `_sysPktBodyCommon`
: The common packet header, as described in _SysPktBodyCommon.

<-- `address`
: The code address whose function name was determined. This is the same `address` that was specified in the request packet.

<-- `startAddr`
: The starting address in target memory of the function that includes the address.

<-- `endAddr` The ending address in target memory of the function that includes the address. If a function name could not be found, this is the last address that was scanned.

<-- `name` The name of the function that includes the `address`. This is a null-terminated string. If a function name could not be found, this is the null string.

Get Trap Breaks

Purpose Retrieves the settings for the trap breaks on the debugging target.

Comments Trap breaks are used to force the debugging target to enter the debugger when a particular system trap is called.

The body of the response packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible trap break.

Each trap break is a single word value that contains the system trap number.

Commands The `Get Trap Breaks` request and response commands are defined as follows:

```
#define sysPktGetTrapBreaksCmd0x10
#define sysPktGetTrapBreaksRsp0x90
```

Request Packet

```
typedef struct SysPktGetTrapBreaksCmdType
{
    _sysPktBodyCommon;
}SysPktGetTrapBreaksCmdType;
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Response Packet

```
typedef struct SysPktGetTrapBreaksRspType
{
    _sysPktBodyCommon;
    Word trapBP[dbgTotalTrapBreaks];
}SysPktGetTrapBreaksRspType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

<-- `trapBP`
An array with an entry for each of the possible trap breaks. A value of `0` indicates that the trap break is not used.

Get Trap Conditionals

Purpose Retrieves the trap conditionals values from the debugging target.

Comments Trap conditionals are used when setting A-Traps for library calls. You can set a separate conditional value for each A-Trap.

The body of the response packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible trap break.

Each trap conditional is a value; if the value of the first word on the stack matches the conditional value when the trap is called, the debugger breaks.

Commands The `Get Trap Conditionals` request and response commands are defined as follows:

```
#define sysPktGetTrapConditionsCmd0x14
#define sysPktGetTrapConditionsRsp0x94
```

Request Packet

```
typedef struct SysPktGetTrapConditionsCmdType
{
```

```
    _sysPktBodyCommon;
}SysPktGetTrapConditionsCmdType
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Response Packet

```
typedef struct SysPktGetTrapConditionsRspType
{
    _sysPktBodyCommon;
    Word trapParam[dbgTotalTrapBreaks];
}SysPktGetTrapConditionsRspType
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

<-- `trapParam` An array with an entry for each of the possible trap breaks. A value of `0` indicates that the trap conditional is not used .

Message

Purpose Sends a message to display on the debugging target.

Comments Application can compile debugger messages into their code by calling the `DbgMessage` function.

The debugging target does not send back a response packet for this command.

Commands The `Message` request command is defined as follows:

```
#define sysPktRemoteMsgCmd0x7F
```

Request Packet

```
typedef struct SysPktRemoteMsgCmdType
```

```
{
    _sysPktBodyCommon;
    Byte text[1];
}SysPktRemoteMsgCmdType;
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `text` .

Read Memory

Purpose Reads memory values from the debugging target.

Comments This command can read up to `sysPktMaxMemChunk` bytes of memory. The actual size of the response packet depends on the number of bytes requested in the request packet.

Commands The `Read Memory` command request and response commands are defined as follows:

```
#define sysPktReadMemCmd0x01
#define sysPktReadMemRsp0x81
```

Request Packet

```
typedef struct SysPktReadMemCmdType
{
    _sysPktBodyCommon;
    void*   address;
    Word    numBytes;
}SysPktReadMemCmdType;
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `address` The address in target memory from which to read values.

--> `numBytes` The number of bytes to read from target memory.

Response Packet

```
typedef struct SysPktReadMemRspType
{
    _sysPktBodyCommon;
    //Byte  data[?];
}SysPktReadMemRspType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

<-- `data` The returned data. The number of bytes in this field matches the `numBytes` value in the request packet.

Read Registers

Purpose Retrieves the value of each of the target processor registers.

Comments The eight data registers are stored in the response packet body sequentially, from `D0` to `D7`. The seven address registers are stored in the response packet body sequentially, from `A0` to `A6`.

Commands The `Read Registers` command request and response commands are defined as follows:

```
#define sysPktReadRegsCmd0x05
#define sysPktReadRegsRsp0x85
```

Request Packet

```
typedef struct SysPktReadRegsCmdType
{
    _sysPktBodyCommon;
```

```
}SysPktReadRegsCmdType;
```

Fields

--> `_sysPktBodyCommon`
: The common packet header, as described in _SysPktBodyCommon.

Response Packet

```
typedef struct SysPktReadRegsRspType
{
    _sysPktBodyCommon;
    M68KRegsType  reg;
}SysPktReadRegsRspType;
```

Fields

<-- `_sysPktBodyCommon`
: The common packet header, as described in _SysPktBodyCommon.

<-- `reg`
: The register values in sequential order: `D0` to `D7`, followed by `A0` to `A6`.

RPC

Purpose Sends a remote procedure call to the debugging target.

Comments .

Commands The `RPC` request and response commands are defined as follows:

```
#define sysPktRPCCmd0x0A
#define sysPktRPCRsp0x8A
```

Request Packet

```
typedef struct SysPktRPCType
{
    _sysPktBodyCommon;
    Word  trapWord;
    DWord resultD0;
```

```
      DWord resultD0;
      Word  numParams;
      SysPktRPCParamTypeparam[?];
}
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `trapWord` The system trap to call.

--> `resultD0` The result from the `D0` register.

--> `resultA0` The result from the `A0` register.

--> `numParams` The number of RPC parameter structures in the `param` array that follows.

--> `param` An array of RPC parameter structures, as described in SysPktRPCParamType.

Set Breakpoints

Purpose Sets breakpoints on the debugging target.

Comments The body of the request packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible breakpoint. If a breakpoint is currently disabled on the debugging target, the enabled field for that breakpoint is set to `0`.

The `dbgTotalBreakpoints` constant is described in Breakpoint Constants.

Commands The `Set Breakpoints` command request and response commands are defined as follows:

```
#define sysPktSetBreakpointsCmd0x0C
#define sysPktSetBreakpointsRsp0x8C
```

Request Packet

```
typedef struct SysPktSetBreakpointsCmdType
{
```

```
    _sysPktBodyCommon;
    BreakpointType db[dbgTotalBreakpoints];
  }SysPktSetBreakpointsCmdType
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `bp`
An array with an entry for each of the possible breakpoints. Each entry is of the type BreakpointType.

Response Packet

```
typedef struct SysPktSetBreakpointsRspType
{
  _sysPktBodyCommon;
}SysPktSetBreakpointsRspType
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Set Trap Breaks

Purpose Sets breakpoints on the debugging target.

Comments The body of the request packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible trap break. If a trap break is currently disabled on the debugging target, the value of that break is set to `0`.

The `dbgTotalBreakpoints` constant is described in Breakpoint Constants.

Commands The `Set Breakpoints` command request and response commands are defined as follows:

```
#define sysPktSetTrapBreaksCmd0x0C
#define sysPktSetTrapBreaksRsp0x8C
```

Request Packet

```
typedef struct SysPktSetTrapBreakssCmdType
{
  _sysPktBodyCommon;
  Word trapBP[dbgTotalBreakpoints];
}SysPktSetTrapBreaksCmdType
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `trapBP` An array with an entry for each of the possible trap breaks. If the value of an entry is 0, the break is not currently in use.

Response Packet

```
typedef struct SysPktSetTrapBreaksRspType
{
  _sysPktBodyCommon;
}SysPktSetTrapBreaksRspType
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Set Trap Conditionals

Purpose Sets the trap conditionals values for the debugging target.

Comments Trap conditionals are used when setting A-Traps for library calls. You can set a separate conditional value for each A-Trap.

The body of the request packet contains an array with `dbgTotalBreakpoints` values in it, one for each possible trap break.

Each trap conditional is a value; if the value of the first word on the stack matches the conditional value when the trap is called, the debugger breaks.

Commands

The `Set Trap Conditionals` request and response commands are defined as follows:

```
#define sysPktSetTrapConditionsCmd0x15
#define sysPktSetTrapConditionsRsp0x95
```

Request Packet

```
typedef struct SysPktSetTrapConditionsCmdType
{
    _sysPktBodyCommon;
    Word trapParam[dbgTotalTrapBreaks];
}SysPktSetTrapConditionsCmdType
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `trapParam`
An array with an entry for each of the possible trap breaks. A value of `0` indicates that the trap conditional is not used .

Response Packet

```
typedef struct SysPktSetTrapConditionsRspType
{
    _sysPktBodyCommon;
}SysPktSetTrapConditionsRspType
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

State

Purpose

Sent by the host program to query the current state of the debugging target, and sent by the target whenever it encounters an exception and enters the debugger.

Comments

The debugging target sends the `State` response packete whenever it enters the debugger for any reason, including a breakpoint, a bus error, a single step, or any other reason.

Commands

The `State` request and response commands are defined as follows:

```
#define sysPktStateCmd0x00
#define sysPktStateRsp0x80
```

Request Packet

```
typedef struct SysPktStateCmdType
{
    _sysPktBodyCommon;
} SysPktStateCmdType
```

Fields

--> `_sysPktBodyCommon`

The common packet header, as described in [_SysPktBodyCommon](#).

Response Packet

```
typedef struct SysPktStateRspType
{
    _sysPktBodyCommon;
    Boolean       resetted;
    Word          exceptionId;
    M68KregsType  reg;
    Word        inst[sysPktStateRspInstWords];
    BreakpointTypebp[dbgTotalBreakpoints];
    void*         startAddr;
    void*         endAddr;
    char          name[sysPktMaxNameLen];
    Byte          trapTableRev;
} SysPktStateRspType;
```

Fields

<-- `_sysPktBodyCommon`	The common packet header, as described in _SysPktBodyCommon.
<-- `resetted`	A Boolean value. This is `TRUE` if the debugging target has just been reset.
<-- `exceptionId`	The ID of the exception that caused the debugger to be entered.
<-- `reg`	The register values in sequential order: `D0` to `D7`, followed by `A0` to `A6`.
<-- `inst`	A buffer of the instructions starting at the current program counter on the debugging target.
<-- `bp`	An array with an entry for each of the possible breakpoints. Each entry is of the type BreakpointType.
<-- `startAddr`	The starting address of the function that generated the exception.
<-- `endAddr`	The ending address of the function that generated the exception.
<-- `name`	The name of the function that generated the exception. This is a null-terminated string. If no name can be found, this is the null string.
<-- `trapTableRev`	The revision number of the trap table on the debugging target. You can use this to determine when the trap table cache on the host computer is invalid.

Toggle Debugger Breaks

Purpose

Enables or disables breakpoints that have been compiled into the code.

Comments

A breakpoint that has been compiled into the code is a special `TRAP` instruction that is generated when source code includes calls to the `DbgBreak` and `DbgSrcBreak` functions.

Sending this command toggles the debugging target between enabling and disabling these breakpoints.

Commands

The `Toggle Debugger Breaks` request and response commands are defined as follows:

```
#define sysPktDbgBreakToggleCmd0x0D
#define sysPktDbgBreakToggleRsp0x8D
```

Request Packet

```
typedef struct SysPktDbgBreakToggleCmdType
{
    _sysPktBodyCommon;
}SysPktDbgBreakToggleCmdType;
```

Fields

--> `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Response Packet

```
typedef struct SysPktDbgBreakToggleRspType
{
    _sysPktBodyCommon;
    Boolean   newState
}SysPktDbgBreakToggleRspType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

<-- `newState` — A Boolean value. If this is set to `TRUE`, the new state has been set to enable breakpoints that were compiled into the code. If this is set to `FALSE`, the new state has been set to disable breakpoints that were compiled into the code.

Write Memory

Purpose Writes memory values to the debugging target.

Comments This command can write up to `sysPktMaxMemChunk` bytes of memory. The actual size of the request packet depends on the number of bytes that you want to write.

Commands The `Write Memory` command request and response commands are defined as follows:

```
#define sysPktWriteMemCmd0x02
#define sysPktWriteMemRsp0x82
```

Request Packet

```
typedef struct SysPktWriteMemCmdType
{
    _sysPktBodyCommon;
    void*   address;
    Word    numBytes;
    //Byte  data[?]
}SysPktWriteMemCmdType;
```

Fields

--> `_sysPktBodyCommon` — The common packet header, as described in _SysPktBodyCommon.

--> `address` — The address in target memory to which the values are written.

--> `numBytes` — The number of bytes to write.

--> `data` — The bytes to write into target memory. The size of this field is defined by the `numBytes` parameter.

Response Packet

```
typedef struct SysPktWriteMemRspType
{
    _sysPktBodyCommon;
}SysPktWriteMemRspType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Write Registers

Purpose Sets the value of each of the target processor registers.

Comments The eight data registers are stored in the request packet body sequentially, from `D0` to `D7`. The seven address registers are stored in the request packet body sequentially, from `A0` to `A6`.

Commands The `Write Registers` command request and response commands are defined as follows:

```
#define sysPktWriteRegsCmd0x06
#define sysPktWriteRegsRsp0x86
```

Request Packet

```
typedef struct SysPktWriteRegsCmdType
{
    _sysPktBodyCommon;
    M68KRegsType  reg;
}SysPktWriteRegsCmdType;
```

Fields

-->`_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

--> `reg` — The new register values in sequential order: `D0` to `D7`, followed by `A0` to `A6`.

Response Packet

```
typedef struct SysPktWriteRegsRspType
{
    _sysPktBodyCommon;
}SysPktWriteRegsRspType;
```

Fields

<-- `_sysPktBodyCommon`
The common packet header, as described in _SysPktBodyCommon.

Summary of Debugger Protocol Packets

Table A.2 summarizes the command packets that you can use with the debugger protocol.

Table A.2 Debugger protocol command packets

Command	Description
Continue	Tells the debugging target to continue execution.
Find	Searches for data in memory on the debugging target.
Get Breakpoints	Retrieves the current breakpoint settings from the debugging target.
Get Routine Name	Determines the name, starting address, and ending address of the function that contains the specified address.
Get Trap Breaks	Retrieves the settings for the trap breaks on the debugging target.
Get Trap Conditionals	Retrieves the trap conditionals values from the debugging target.

Table A.2 Debugger protocol command packets *(continued)*

Command	Description
Message	Sends a message to display on the debugging target.
Read Memory	Reads memory values from the debugging target.
Read Registers	Retrieves the value of each of the target processor registers.
RPC	Sends a remote procedure call to the debugging target.
Set Breakpoints	Sets breakpoints on the debugging target.
Set Trap Breaks	Sets breakpoints on the debugging target.
Set Trap Conditionals	Sets the trap conditionals values for the debugging target.
State	Sent by the host program to query the current state of the debugging target, and sent by the target whenever it encounters an exception and enters the debugger.
Toggle Debugger Breaks	Enables or disables breakpoints that have been compiled into the code.
Write Memory	Writes memory values to the debugging target.
Write Registers	Sets the value of each of the target processor registers.

B

Host Control API

This appendix describes the host control API, which you can use the to call emulator-defined functions while your application is running under the Palm OS® Emulator. For example, you can make function calls to start and stop profiling in the emulator.

IMPORTANT: This chapter describes the version of the host control API that shipped on the Metrowerks CodeWarrior for the Palm Operating System, Version 6 CD-ROM. If you are using a different version, the features in your version might be different than the features described here.

The host control functions are defined in the `HostControl.h` header file. These functions are invoked by executing a trap / selector combination that is defined for use by the emulator and other foreign host environments. Palm OS Emulator catches the calls intended for it that are made to this selector.

Constants

This section lists the constants that you use with the host control API.

Host Error Constants

Several of the host control API functions return a `HostErr` value.

```
enum
{
hostErrNone = 0,
hostErrUnknownGestaltSelector,
hostErrDiskError,
hostErrOutOfMemory,
hostErrMemReadOutOfRange,
```

```
hostErrMemWriteOutOfRange,
hostErrMemInvalidPtr,
hostErrInvalidParameter,
hostErrTimeout,
hostErrInvalidDeviceType,
hostErrInvalidRAMSize,
hostErrFileNotFound,
hostErrRPCCall,
hostErrSessionRunning,
hostErrSessionNotRunning,
hostErrNoSignalWaiters,
hostErrSessionNotPaused
};
```

`hostErrNone` No error.

`hostErrUnknownGestaltSelector`
The specified Gestalt selector value is not valid.

`hostErrDiskError`
A disk error occurred.

`hostErrOutOfMemory`
There is not enough memory to complete the request.

`hostErrMemReadOutOfRange`
An out of range error occurred during a memory read.

`hostErrMemWriteOutOfRange`
An out of range error occurred during a memory write.

`hostErrMemInvalidPtr`
The pointer is not valid.

`hostErrInvalidParameter`
A parameter to a function is not valid.

`hostErrTimeout`
A timeout occurred.

`hostErrInvalidDeviceType`
The specified device type is not valid.

`hostErrInvalidRAMSize`
The specified RAM size value is not valid.

`hostErrFileNotFound`
The specified file could not be found.

`hostErrRPCCall`
A function that must be called remotely was called by an application. These functions include: `HostSessionCreate`, `HostSessionOpen`, `HostSessionClose`, `HostSessionQuit`, `HostSignalWait`, and `HostSignalResume`.

`hostErrSessionRunning`
A session is already running and one of the following functions was called: `HostSessionCreate`, `HostSessionOpen`, or `HostSessionQuit`.

`hostErrSessionNotRunning`
No session is running and the `HostSessionClose` function was called.

`hostErrNoSignalWaiters`
The `HostSendSignal` function was called, but there are no external scripts waiting for a signal.

`hostErrSessionNotPaused`
The `HostSignalResume` function was called, but the session has not been paused by a call to `HostSendSignal`.

Host Function Selector Constants

You can use the host function selector constants with the HostIsSelectorImplemented function to determine if a certain function is implemented on your debugging host. Each constant is the name of a function, with the `Host` portion replaced by `HostSelector`.

```
typedef enum
{
```

```
hostSelectorGetHostVersion,
hostSelectorGetHostID,
hostSelectorGetHostPlatform,
hostSelectorIsSelectorImplemented,
hostSelectorGestalt,
hostSelectorIsCallingTrap,
hostSelectorProfileInit,
hostSelectorProfileStart,
hostSelectorProfileStop,
hostSelectorProfileDump,
hostSelectorProfileCleanup,
hostSelectorProfileDetailFn,
hostSelectorErrNo,
hostSelectorFClose,
hostSelectorFEOF,
hostSelectorFError,
hostSelectorFFlush,
hostSelectorFGetC,
hostSelectorFGetPos,
hostSelectorFGetS,
hostSelectorFOpen,
hostSelectorFPrintF,
hostSelectorFPutC,
hostSelectorFPutS,
hostSelectorFRead,
hostSelectorRemove,
hostSelectorRename,
hostSelectorFReopen,
hostSelectorFScanF,
hostSelectorFSeek,
hostSelectorFSetPos,
hostSelectorFTell,
hostSelectorFWrite,
hostSelectorTmpFile,
hostSelectorTmpNam,
hostSelectorGetEnv,
hostSelectorMalloc,
hostSelectorRealloc,
hostSelectorFree,
hostSelectorGremlinIsRunning,
```

```
  hostSelectorGremlinNumber,
  hostSelectorGremlinCounter,
  hostSelectorGremlinLimit,
  hostSelectorGremlinNew,
  hostSelectorImportFile,
  hostSelectorExportFile,
  hostSelectorGetPreference),
  hostSelectorSetPreference,
  hostSelectorLogFile,
  hostSelectorSetLogFileSize,
  hostSelectorSessionCreate,
  hostSelectorSessionOpen,
  hostSelectorSessionClose,
  hostSelectorSessionQuit,
  hostSelectorSignalSend,
  hostSelectorSignalWait,
  hostSelectorSignalResume,
  hostSelectorTraceInit,
  hostSelectorTraceClose,
  hostSelectorTraceOutputT,
  hostSelectorTraceOutputTL,
  hostSelectorTraceOutputVT,
  hostSelectorTraceOutputVTL,
  hostSelectorTraceOutputB,
  hostSelectorLastTrapNumber
} HostControlTrapNumber;
```

Host ID Constants

Some of the host control API functions use a Host ID value to specify the debugging host type.

```
enum
{
hostIDPalmOS,
hostIDPalmOSEmulator,
hostIDPalmOSSimulator
};
```

`hostIDPalmOS` A Palm Computing Platform hardware device.

`hostIDPalmOSEmulator`
The Palm OS Emulator application.

`hostIDPalmOSSimulator`
The Macintosh Simulator application.

Host Platform Constants

Several of the host control API functions use a `HostPlatform` value to specify operating system hosting the emulation.

```
enum
{
  hostPlatformPalmOS,
  hostPlatformWindows,
  hostPlatformMacintosh,
  hostPlatformUnix
};
```

`hostPlatformPalmOS`
The Palm OS platform.

`hostPlatformWindows`
The Windows operating system platform.

`hostPlatformMacintosh`
The Mac OS platform.

`hostPlatformUnix`
The Unix operating system platform.

Host Signal Constants

This section describes the host signal values, which you can use with the HostSendSignal.

```
enum
{
  hostSignalReserved,
  hostSignalIdle,
  hostSignalQuit,
  hostSignalSessionStarted,
```

```
    hostSignalSessionStopped,
    hostSignalGremlinStarted,
    hostSignalGremlinSuspended,
    hostSignalGremlinResumed,
    hostSignalGremlinStopped,
    hostSignalHordeStopped,
    hostSignalUser
  };
```

`hostSignalReserved`
: System-defined signals start here.

`hostSignalIdle`
: Palm OS Emulator is about to go into an idle state.

`hostSignalQuit`
: Palm OS Emulator is about to quit.

`hostSignalSessionStarted`
: **Not Yet Implemented**.

`hostSignalSessionStopped`
: **Not Yet Implemented**.

`hostSignalGremlinStarted`
: **Not Yet Implemented**.

`hostSignalGremlinSuspended`
: **Not Yet Implemented**.

`hostSignalGremlinResumed`
: **Not Yet Implemented**.

`hostSignalGremlinStopped`
: **Not Yet Implemented**.

`hostSignalHordeStopped`
: **Not Yet Implemented**.

`hostSignalUser`
: User-defined signals start at this value.

Data Types

This section describes the data types that you use with the host control API.

HostFILE

The host control file operations create and use the `HostFile` data structure for file access.

```
typedef struct HostFILE
{
  long _field;
} HostFILE;
```

HostBool

The host control API defines `HostBool` for use as a Boolean value.

```
typedef long HostBool;
```

HostGremlinInfo

The host control API defines the `HostGremlinInfo` structure type to store information about a horde of gremlins.

```
typedef struct HostGremlinInfo
{
  long  fFirstGremlin;
  long  fLastGremlin;
  long  fSaveFrequency;
  long  fSwitchDepth;
  long  fMaxDepth;
  long  fAppNames[200];
};

typedef struct HostGremlinInfo HostGremlinInfo;
```

HostGremlinInfo Fields

`fFirstGremlin`	The number of the first gremlin to run.
`fLastGremlin`	The number of the last gremlin to run.

`fSaveFrequency`	The gremlin snapshot frequency.
`fSwitchDepth`	The number of gremlin events to generate before switching to another gremlin.
`fMaxDepth`	The maximum number of gremlin events to generate for each gremlin.
`fAppNames`	A comma-separated string containing a list of application names among which the gremlin horde is allowed to switch. If this string is empty, all applications are available for use with the gremlins. If this string begins with a dash (`'-'`), the applications named in the string are excluded, rather than included in the list of available applications.

HostID

The host control API defines `HostID` for use as an identifier value.

```
typedef long HostID;
```

HostPlatform

The host control API defines `HostPlatform` for use as a platform identifier value.

```
typedef long HostPlatform;
```

HostSignal

The host control API defines `HostSignal` for use as a platform identifier value.

```
typedef long HostSignal;
```

Functions

This section describes the host control API functions.

HostErrNo

Purpose To return the value of errNO, the standard C library variable that reflects the result of many standard C library functions. You can call this function after calling one of the Host Control functions that wraps the standard C library.

IMPORTANT: The HostErrNo function is only applicable to functions that wrap the standard C library. It is not applicable to all Host Control functions.

Prototype

```
long HostErrNo(void);
```

Parameters None.

Result The error number.

HostExportFile

Purpose Copies a database from the handheld device to the desktop computer.

Prototype

```
HostErr HostExportFile(const char* fileName,
long cardNum, const char* dbName);
```

Parameters

`fileName`	The file name to use on the desktop computer.
`cardNum`	The number of the card on the handheld device on which the database is contained.
`dbName`	The name of the handheld database.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFClose

Purpose Closes a file on the desktop computer.

Prototype `long HostFClose(HostFILE* f);`

Parameters `f` The file to close.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFEOF

Purpose Determines if the specified file is at its end.

Prototype `long HostFEOF(HostFILE* f);`

Parameters `f` The file to test.

Result Returns `0` if the specified file is at its end, and a non-zero value otherwise.

HostFError

Purpose Retrieves the error code from the most recent operation on the specified file.

Prototype `long HostFError(HostFILE* f);`

Parameters `f` The file.

Result The error code from the most recent operation on the specified file. Returns `0` if no errors have occurred on the file.

HostFFlush

Purpose Flushes the buffer for the specified file.

Prototype `long HostFFlush(HostFILE* f);`

Parameters

`f`	The file to flush.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFGetC

Purpose Retrieves the character at the current position in the specified file.

Prototype `long HostFGetC(HostFILE* f);`

Parameters

`f`	The file.

Result The character, or `EOF` to indicate an error.

HostFGetPos

Purpose Retrieves the current position in the specified file.

Prototype `long HostFGetPos(HostFILE* f, long* posn);`

Parameters

`f`	The file.
`posn`	Upon successful return, the current position in the file.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFGetS

Purpose Retrieves a character string from the selected file and returns a pointer to that string.

Prototype

```
char* HostFGetS(char* s, long n, HostFILE* f);
```

Parameters

`s`	A pointer to the string buffer to be filled with characters from the file.
`n`	The number of characters to retrieve.
`f`	The file.

Result The character string, or `NULL` to indicate an error.

HostFOpen

Purpose Opens a file on the desktop computer.

Prototype

```
HostFILE* HostFOpen(const char* name,
const char* mode);
```

Parameters

`name`	The name of the file to open.
`mode`	The mode to use when opening the file.

Result The file stream pointer, or `NULL` to indicate an error.

HostFPrintF

Purpose Writes a formatted string to a file.

Prototype

```
long HostFPrintF(HostFILE* f, const char* format,
...);
```

Parameters

`f`	The file to which the string is written.
`format`	The format specification.

	`...`	String arguments.

Result The number of characters actually written.

HostFPutC

Purpose Writes a character to the specified file.

Prototype

```
long HostFPutC(long c, HostFILE* f);
```

Parameters

`c`	The character to write.
`f`	The file to which the character is written.

Result The number of characters written, or `EOF` to indicate an error.

HostFPutS

Purpose Writes a string to the specified file.

Prototype

```
long HostFPutS(const char* s, HostFILE* f);
```

Parameters

`s`	The string to write.
`f`	The file to which the character is written.

Result A non-negative value if the operation was successful, or a negative value to indicate failure.

HostFRead

Purpose Reads a number of items from the file into a buffer.

Prototype

```
long HostFRead(void* buffer, long size,
long count, HostFILE* f);
```

Parameters

`buffer`	The buffer into which data is read.
`size`	The size of each item.

`count`	The number of items to read.
`f`	The file from which to read.

Result The number of items that were actually read.

HostFree

Purpose Frees memory on the desktop computer.

Prototype

```
void HostFree(void* p);
```

Parameters

`p`	A pointer to the memory block to be freed.

Result None.

HostFReopen

Purpose Changes the file with which the stream `f` is associated. `HostFReopen` first closes the file that was associated with the stream, then opens the new file and associates it with the same stream.

Prototype

```
HostFILE* HostFReopen(const char* name,
const char* mode, HostFILE *f);
```

Parameters

`name`	The name of the file to open.
`mode`	The mode to use when opening the file.
`f`	The file from which to read.

Result The file stream pointer, or `NULL` to indicate an error.

HostFScanF

Purpose Reads formatted text from a file.

Prototype
```
long HostFReopen(HostFILE* f, const char *fmt,
...);
```

Parameters

`f`	The file from which to read input.
`fmt`	A format string, as used in standard C-library calls such as `scanf`.
...	The list of variables into which scanned input are written.

Result The number of items that were read, or a negative value to indicate an error.

Returns `EOF` if end of file was reached while scanning.

HostFSeek

Purpose Moves the file pointer to the specified position.

Prototype
```
long HostFSeek(HostFILE* f, long offset,
long origin);
```

Parameters

`f`	The file.
`offset`	The number of bytes to move from the initial position, which is specified in the `origin` parameter.
`origin`	The initial position.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFSetPos

Purpose Sets the position indicator of the file.

Prototype

```
long HostFSetPos(HostFILE* f, long posn);
```

Parameters

`f`	The file.
`posn`	The position value.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostFTell

Purpose Retrieves the current position of the specified file.

Prototype

```
long HostFTell(HostFILE* f);
```

Parameters

`f`	The file.

Result Returns `-1` to indicate an error.

HostFWrite

Purpose Writes data to a file.

Prototype

```
long HostFWrite(const void* buffer, long size,
long count, HostFILE* f);
```

Parameters

`buffer`	The buffer that contains the data to be written.
`size`	The size of each item.
`count`	The number of items to write.
`f`	The file to which the data is written.

Result The number of items actually written.

HostGetEnv

Purpose Retrieves the value of an environment variable.

Prototype `char* HostGetEnv(char* varName);`

Parameters

`varName`	The name of the environment variable that you want to retrieve.

Result The string value of the named variable, or `NULL` if the variable cannot be retrieved.

HostGestalt

Purpose Currently does nothing except return an “invalid selector” error. In the future, this function will be used for queries about the runtime environment.

Prototype `HostErr HostGestalt(long gestSel, long* response);`

Parameters

HostGetHostID

Purpose Retrieves the ID of the debugging host. This is one of the constants described in Host ID Constants. Palm OS Emulator always returns the value `hostIDPalmOSEmulator`.

Prototype `HostID HostGetHostID(void);`

Parameters None.

Result The host ID.

HostGetHostPlatform

Purpose Retrieves the host platform ID, which is one of the values described in Host Platform Constants.

Prototype `HostPlatform HostGetHostPlatform(void);`

Parameters None.

Result The platform ID.

HostGetHostVersion

Purpose Retrives the version number of the debugging host.

Prototype `long HostGetHostVersion(void);`

Parameters None.

Result The version number.

Comments This function returns the version number in the same format that is used by the Palm OS, which means that you can access the version number components using the following macros from the `SystemMgr.h` file:

```
sysGetROMVerMajor(dwROMVer)
sysGetROMVerMinor(dwROMVer)
sysGetROMVerFix(dwROMVer)
sysGetROMVerStage(dwROMVer)
sysGetROMVerBuild(dwROMVer)
```

HostGetPreference

Purpose Retrieves the specified preference value.

Prototype

```
HostBool HostGetPreference(const char* prefName,
char* prefValue);
```

Parameters

`prefName`	The name of the preference whose value you want to retrieve.
`prefValue`	Upon successful return, the string value of the specified preference.

Result A Boolean value that indicates whether the preference was successfully retrieved.

Comments Each preference is identified by name. You can view the preference names in the Palm OS Emulator preferences file for your platform, as shown in Table B.1.

Table B.1 Palm OS Emulator preferences file names and locations

Platform	File name	File location
Macintosh	`Palm OS Emulator Preferences`	In the `Preferences` folder.
Windows	`Palm OS Emulator Preferences.ini`	In the `Windows System` directory.
Unix	`.poserrc`	In your home directory.

See Also The HostSetPreference function.

HostGremlinCounter

Purpose Returns the current event count of the currently running gremlin.

Prototype `long HostGremlinCounter(void);`

Parameters None.

Result The event count.

Comments This return value of this function is only valid if a gremlin is currently running.

HostGremlinIsRunning

Purpose Determines if a gremlin is currently running.

Prototype `HostBool HostGremlinIsRunning(void);`

Parameters None.

Result A Boolean value indicating whether a gremlin is currently running.

HostGremlinLimit

Purpose Retrieves the limit value of the currently running gremlin.

Prototype `long HostGremlinLimit(void);`

Parameters None.

Result The limit value of the currently running gremlin.

Comments This return value of this function is only valid if a gremlin is currently running.

HostGremlinNew

Purpose Creates a new gremlin.

Prototype
```
HostErr HostGremlinNew(
const HostGremlinInfo* info);
```

Parameters TBD.

HostGremlinNumber

Purpose Retrieves the number of the currently running gremlin.

Prototype
```
long HostGremlinNumber(void);
```

Parameters None.

Result The gremlin number of the currently running gremlin.

Comments This return value of this function is only valid if a gremlin is currently running.

HostImportFile

Purpose Copies a database from the desktop computer to the handheld device, and stores it on the specified card number. The database name on the handheld device is the name stored in the file.

Prototype
```
HostErr HostImportFile(const char* fileName,
long cardNum);
```

Parameters

`fileName`	The file on the desktop computer that contains the database.

`cardNum`	The card number on which the database is to be installed. You almost always use `0` to specify the built-in RAM.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostLogFile

Purpose Returns a reference to the file that the Emulator is using to log information. You can use this to add your own information to the same file.

Prototype `HostFILE* HostLogFile(void);`

Parameters None.

Result A pointer to the log file, or `NULL` if not successful.

HostIsCallingTrap

Purpose Determines if Palm OS Emulator is currently calling a trap.

Prototype `HostBool HostIsCallingTrap(void);`

Parameters None.

Result TRUE if Palm OS Emulator is currently calling a trap, and FALSE if not.

HostIsSelectorImplemented

Purpose Determines if the specified function selector is implemented on the debugging host.

Prototype `HostBool HostIsSelectorImplemented(long selector);`

Parameters `selector` The function selector. This must be one of the constants described in Host Function Selector Constants.

Result TRUE if the specified function selector is implemented on the host, and FALSE if not.

HostMalloc

Purpose Allocates a memory block on the debugging host.

Prototype `void* HostMalloc(long size);`

Parameters `size` The number of bytes to allocate.

Result A pointer to the allocated memory block, or `NULL` if there is not enough memory available.

HostProfileCleanup

Purpose Releases the memory used for profiling and disables profiling.

Prototype `HostErr HostProfileCleanup(void);`

Parameters None.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostProfileDetailFn

Purpose Profiles the function that contains the specified address.

Prototype

```
HostErr HostProfileDetailFn(void* addr,
HostBool logDetails);
```

Parameters

`addr`	The address in which you are interested.
`logDetails`	A Booleavn value. If this is TRUE, profiling is performed at a machine-language instruction level, which means that each opcode is treated as its own function.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostProfileDump

Purpose Writes the current profiling information to the named file.

Prototype

```
HostErr HostProfileDump(const char* filename);
```

Parameters

`filename`	The name of the file to which the profile information gets written.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostProfileInit

Purpose Initializes and enables profiling in the debugging host.

Prototype

```
HostErr HostProfileInit(long maxCalls,
long maxDepth);
```

Parameters

`maxCalls`	The maximum number of calls to profile.

`maxDepth`	The maximum profiling depth.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostProfileStart

Purpose Turns profiling on.

Prototype `HostErr HostProfileStart(void);`

Parameters None.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostProfileStop

Purpose Turns profiling off.

Prototype `HostErr HostProfileStop(void);`

Parameters None.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostRealloc

Purpose Reallocates space for the specified memory block.

Prototype `void* HostRealloc(void* ptr, long size);`

Parameters

`ptr`	A pointer to a memory block that is being resized.

`size`	The new size for the memory block.

Result A pointer to the allocated memory block, or `NULL` if there is not enough memory available.

HostRemove

Purpose Deletes a file.

Prototype

```
long HostRemove(const char* name);
```

Parameters

`name`	The name of the file to be deleted.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostRename

Purpose Renames a file.

Prototype

```
long HostRemove(const char* oldName,
const char* newName);
```

Parameters

`oldName`	The name of the file to be renamed.
`newName`	The new name of the file.

Result Returns `0` if the operation was successful, and a non-zero value if not.

HostSessionClose

Purpose Closes the current emulation session.

Prototype

```
HostErr HostSessionClose(const char* psfFileName);
```

Parameters

`psfFileName`	The name of the file to which the current session is to be saved.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments This function is defined for external RPC clients to call; the effect of calling it for Palm OS applications running on the emulated device is undefined.

HostSessionCreate

Purpose Creates a new emulation session.

Prototype

```
HostErr HostSessionCreate(const char* device,
long ramSize, const char* romPath);
```

Parameters

`device`	The name of the handheld device to emulate in the session.
`ramSize`	The amount of emulated RAM in the new session.
`romPath`	The path to the ROM file for the new session.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments This function is defined for external RPC clients to call; the effect of calling it for Palm OS applications running on the emulated device is undefined.

WARNING! This function is not implemented in the current version of Palm OS Emulator; however, it will be implemented in the near future.

HostSessionOpen

Purpose Opens a previously saved emulation session.

Prototype `HostErr HostSessionOpen(const char* psfFileName);`

Parameters `psfFileName` The name of the file containing the saved session that you want to open.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments This function is defined for external RPC clients to call; the effect of calling it for Palm OS applications running on the emulated device is undefined.

WARNING! This function is not implemented in the current version of Palm OS Emulator; however, it will be implemented in the near future.

HostSessionQuit

Purpose Asks Palm OS Emulator to quit. Returns an error if a session is already running.

Prototype `HostErr HostSessionQuit(void);`

Parameters None.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments This function is defined for external RPC clients to call; the effect of calling it for Palm OS applications running on the emulated device is undefined.

IMPORTANT: This function is defined for external RPC clients to call, and returns an error if you call it from within a Palm application.

HostSetLogFileSize

Purpose Determines the size of the logging file that Palm OS Emulator is using.

Prototype

```
void HostSetLogFileSize(long size);
```

Parameters

`size`	The new size for the logging file, in bytes.

Result None.

Comments By default, Palm OS Emulator saves the last 1 megabyte of log data to prevent logging files from becoming enormous. You can call this function to change the log file size.

HostSetPreference

Purpose Sets the specified preference value.

Prototype

```
void HostSetPreference(const char* prefName,
const char* prefValue);
```

Parameters

`prefName`	The name of the preference whose value you are setting.
`prefValue`	The new value of the preference.

Result None.

Comments Each preference is identified by name. You can view the preference names in the Palm OS Emulator preferences file for your platform, as shown in Table B.1.

See Also The HostGetPreference function.

HostSignalResume

Purpose Restarts Palm OS Emulator after it has issued a signal.

Prototype `HostErr HostSignalResume(void);`

Parameters None.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments Palm OS Emulator waits to be restarted after issuing a signal to allow external scripts to perform operations.

See Also The HostSignalSend and HostSignalWait functions.

IMPORTANT: This function is defined for external RPC clients to call, and returns an error if you call it from within a Palm application.

HostSignalSend

Purpose Sends a signal to any scripts that have HostSignalWait calls pending.

Prototype `HostErr HostSignalSend(HostSignal signalNumber);`

Parameters

`signalNumber`	The signal for which you want to wait. This can be a predefined signal or one that you have defined.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments Palm OS Emulator halts and waits to be restarted after sending the signal. This allows external scripts to perform operations. The external script must call the HostSignalResume function to restart POSE.

If there are not any scripts waiting for a signal, Palm OS Emulator does not halt.

The predefined signals are:

- `hostSignalIdle`, which Palm OS Emulator issues when it detects that it is going into an idle state.
- `hostSignalQuit`, which Palm OS Emulator issues when it is about to quit.

See Also The HostSignalResume and HostSignalWait functions.

HostSignalWait

Purpose Waits for a signal from POSE, and returns the signalled value.

Prototype
```
HostErr HostSignalWait(long timeout,
HostSignal* signalNumber);
```

Parameters

`timeout`	The number of milliseconds to wait for the signal before timing out.

`signalNumber`	The signal for which you want to wait. This can be a predefined signal or one that you have defined.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments The predefined signals are:

- `hostSignalIdle`, which Palm OS Emulator issues when it detects that it is going into an idle state.
- `hostSignalQuit`, which Palm OS Emulator issues when it is about to quit.

See Also The HostSignalResume and HostSignalSend functions.

IMPORTANT: This function is defined for external RPC clients to call, and returns an error if you call it from within a Palm application.

HostTmpFile

Purpose Returns the temporary file used by the debugging host.

Prototype `HostFILE* HostTmpFile(void);`

Parameters None.

Result A pointer to the temporary file, or `NULL` if an error occurred.

HostTmpNam

Purpose Creates a unique temporary file name.

Prototype `char* HostTmpNam(char* s);`

Parameters

`s`	Either be a `NULL` pointer or a pointer to a character array. The character array must be at least `L_tmpnam` characters long. If `s` is not NULL, the newly created temporary file name is stored into `s`.

Result A pointer to an internal static object that the calling program can modify.

HostTraceInit

Purpose Initiates a connection to the external trace reporting tool.

Prototype `void HostTraceInit(void);`

Parameters None.

NOTE: The tracing functions are used in conjunction with an external trace reporting tool. You can call these functions to send information to the external tool in real time.

Result None.

HostTraceClose

Purpose Closes the connection to the external trace reporting tool.

Prototype

```
void HostTraceClose(void);
```

Parameters None.

Result None.

HostTraceOutputT

Purpose Outputs a text string to the external trace reporting tool.

Prototype

```
void HostTraceOutputT(unsigned short moduleId,
const char* fmt, ...);
```

Parameters

`moduleId`	The ID of the Palm OS subsystem from which this output originates. You can use this with the external tracing tool to filter traces according to their origin. The ID must match one of the error classes defined in the `SystemMgr.h` file.
`fmt`	A format string, as used in standard C-library calls such as `printf`. The format string has the following form: `% flags width type`
...	The list of variables to be formatted for output.

Table B.2 shows the flag types that you can use in the format specification for the tracing output functions.

Table B.2 Trace function format specification flags

Flag	Description
`-`	Left-justified output.
`+`	Always display the sign symbol.
`space`	Display a space when the value is positive, rather than a '+' symbol.
`#`	Alternate form specifier.

Table B.3 shows the output types that you can use in the format specification for the tracing output functions.

Table B.3 Trace function format specification types

Flag	Description
`%`	Displays the '%' character.
`s`	Displays a null-terminated string value.
`c`	Displays a character value.
`ld`	Displays an Int32 value.
`lu`	Displays a UInt32 value.
`lx` or `lX`	Displays a UInt32 value in hexadecimal.
`hd`	Displays an Int16 value.
`hu`	Displays a UInt16 value.
`hx` or `hX`	Displays an Int16 or UInt16 value i hexadecimal.

Result None.

HostTraceOutputTL

Purpose Outputs a text string, followed by a newline, to the external trace reporting tool. This function performs the same operation as the `HostTraceOutputT` function, and adds the newline character.

Prototype

```
voidHostTraceOutputTL(unsigned short moduleId,
const char* fmt, ...);
```

Parameters

`moduleId`	The ID of the Palm OS subsystem from which this output originates. You can use this with the external tracing tool to filter traces according to their origin. The ID must match one of the error classes defined in the `SystemMgr.h` file.
`fmt`	A format string, as used in standard C-library calls such as `printf`. For more information about the formatting specification, see the description of the HostTraceOutputT function.
...	The list of variables to be formatted for output.

Result None.

HostTraceOutputVT

Purpose Outputs a text string to the external trace reporting tool.

Prototype

```
void HostTraceOutputVT(unsigned short moduleId,
const char* fmt, va_list vargs);
```

Parameters

`moduleId`	The ID of the Palm OS subsystem from which this output originates. You can use this with the external tracing tool to filter traces according to their origin. The ID must match one of the error classes defined in the `SystemMgr.h` file.

`fmt`	A format string, as used in standard C-library calls such as `printf`. For more information about the formatting specification, see the description of the HostTraceOutputT function.
`vargs`	A structure containing the variable argument list. This is the same kind of variable argument list used for standard C-library functions such as `vprintf`.

Result None.

HostTraceOutputVTL

Purpose Outputs a text string, followed by a newline, to the external trace reporting tool. This function performs the same operation as the `HostTraceOutputVT` function, and adds the newline character.

Prototype

```
void HostTraceOutputVTL(unsigned short moduleId,
const char* fmt, va_list vargs);
```

Parameters

`moduleId`	The ID of the Palm OS subsystem from which this output originates. You can use this with the external tracing tool to filter traces according to their origin. The ID must match one of the error classes defined in the `SystemMgr.h` file.
`fmt`	A format string, as used in standard C-library calls such as `printf`. For more information about the formatting specification, see the description of the HostTraceOutputT function.
`vargs`	A structure containing the variable argument list. This is the same kind of variable argument list used for standard C-library functions such as `vprintf`.

Result None.

HostTraceOutputB

Purpose Outputs a buffer of data, in hex dump format, to the external trace reporting tool.

Prototype

```
void HostTraceOutputB(unsigned short moduleId,
const unsigned char* buffer,
unsigned long len/*size_t*/);
```

Parameters

`moduleId`	The ID of the Palm OS subsystem from which this output originates. You can use this with the external tracing tool to filter traces according to their origin. The ID must match one of the error classes defined in the `SystemMgr.h` file.
`buffer`	A pointer to a buffer of raw data.
`len`	The number of bytes of data in the buffer.

Result None.

Summary of Host Control API Functions

The tables in this section summarize the host control API functions.

Host Control Database Functions

Table B.4 Host control database functions

Function	Description
`HostExportFile`	Copies a database from the handheld device to the desktop computer.
`HostImportFile`	Copies a database from the desktop computer to the handheld device, and stores it on the specified card number. The database name on the handheld device is the name stored in the file.

Host Control Environment Functions

Table B.5 Host control environment functions

Function	Description
HostGestalt	Currently does nothing except to return an "invalid selector" error.
HostGetHostID	Retrieves the ID of the debugging host. Palm OS Emulator always returns the value `hostIDPalmOSEmulator`.
HostGetHostPlatform	Retrieves the host platform ID.
HostGetHostVersion	Returns the version number of the debugging host.
HostIsCallingTrap	Returns a Boolean indicating whether the specified function selector is implemented on the debugging host.
HostIsSelectorImplemented	Returns a Boolean indicating whether the specified function selector is implemented on the debugging host.

Host Control Gremlin Functions

Table B.6 Host control gremlin functions

Function	Description
HostGremlinCounter	Returns the current count for the currently running gremlin.
HostGremlinIsRunning	Returns a Boolean value indicating whether a gremlin is currently running.
HostGremlinLimit	Returns the limit value of the currently running gremlin.
HostGremlinNew	Creates a new gremlin.
HostGremlinNumber	Returns the gremlin number of the currently running gremlin.

Host Control Logging Functions

Table B.7 Host control preference functions

Function	Description
HostLogFile	Returns a reference to the file that Palm OS Emulator is using to log information.
HostSetLogFileSize	Modifies the size of the logging file.

Host Control Preference Functions

Table B.8 Host control preference functions

Function	Description
HostGetPreference	Retrieves the value of a preference.
HostSetPreference	Sets a new value for a preference.

Host Control Profiling Functions

Table B.9 Host control profiling functions

Function	Description
HostProfileCleanup	Releases the memory used for profiling and disables profiling.
HostProfileDetailFn	Profiles the function that contains the specified address.
HostProfileDump	Writes the current profiling information to the named file.
HostProfileInit	Initializes and enables profiling in the debugging host.
HostProfileStart	Turns profiling on.
HostProfileStop	Turns profiling off.

Host Control RPC Functions

Table B.10 Host control RPC functions

Function	Description
HostRename	Closes the currently executing emulation session.
HostSessionClose	Closes the current emulation session
HostSessionCreate	Creates a new emulation session.
HostSessionOpen	Opens a previously saved emulation session.
HostSessionQuit	Asks Palm OS Emulator to quit.
HostSignalResume	Resumes Palm OS Emulator after it has halted to wait for external scripts to handle a signal.
HostSignalSend	Sends a signal to external scripts.
HostSignalWait	Waits for Palm OS Emulator to send a signal.

Host Control Standard C-Library Functions

Table B.11 Host control standard C-library functions

Function	Description
HostErrNo	Returns the error number from the most recent host control API operation.
HostFClose	Closes a file on the desktop computer. Returns `0` if the operation was successful, and a non-zero value if not.
HostFEOF	Returns `0` if the specified file is at its end, and a non-zero value otherwise.
HostFError	Returns the error code from the most recent operation on the specified file. Returns `0` if no errors have occurred on the file.
HostFFlush	Flushes the buffer for the specified file.

Table B.11 Host control standard C-library functions

Function	Description
HostFGetC	Returns the character at the current position in the specified file. Returns `EOF` to indicate an error.
HostFGetPos	Retrieves the current position in the specified file. Returns `0` if the operation was successful, and a non-zero value if not.
HostFGetS	Retrieves a character string from the selected file and returns a pointer to that string. Returns `NULL` to indicate an error.
HostFOpen	Opens a file on the desktop computer and returns a `HostFILE` pointer for that file. Returns `NULL` to indicate an error.
HostFPrintF	Writes a formatted string to a file, and returns the number of characters written.
HostFPutC	Writes a character to the specified file, and returns the character written. Returns `EOF` to indicate an error.
HostFPutS	Writes a string to the specified file, and returns a non-negative value to indicate success.
HostFRead	Reads a number of items from the file into a buffer. Returns the number of items that were actually read.
HostFree	Frees memory on the desktop computer.
HostFReopen	Associates a file stream with a different file.
HostFScanF	Scans a file for formatted input.
HostFSeek	Moves the file pointer to the specified position, and returns `0` to indicate success.
HostFSetPos	Sets the position indicator of the file, and returns `0` to indicate success.

Table B.11 Host control standard C-library functions

Function	Description
`HostFTell`	Retrieves the current position of the specified file. Returns `-1` to indicate an error.
`HostFWrite`	Writes data to a file, and returns the actual number of items written.
`HostGetEnv`	Retrieves the value of an environment variable.
`HostMalloc`	Allocates a memory block on the debugging host, and returns a pointer to the allocated memory. Returns NULL if there is not enough memory available.
`HostRealloc`	Reallocates space for the specified memory block.
`HostRemove`	Deletes a file.
`HostRename`	Renames a file.
`HostTmpFile`	Returns the temporary file used by the debugging host.
`HostTmpNam`	Creates a unique temporary file name.

Host Control Tracing Functions

Table B.12 Host control tracing functions

Function	Description
`HostTraceInit`	Must be called before logging any trace information.
`HostTraceClose`	Must be called when done logging trace information.
`HostTraceOutputT`	Outputs text to the trace log using printf-style formatting.

Table B.12 Host control tracing functions *(continued)*

Function	Description
`HostTraceOutputTL`	Outputs text to the trace log using printf-style formatting, and appends a newline character to the text.
`HostTraceOutputVT`	Outputs text to the trace log using vprintf-style formatting.
`HostTraceOutputVTL`	Outputs text to the trace log using vprintf-style formatting, and appends a newline character to the text.
`HostTraceOutputB`	Outputs a buffer of raw data to the trace log in hex dump format.

C

Simple Data Types

Table C.1 describes the simple data types, which have been renamed in the newest release of the Palm OS® software.

Table C.1 Simple Data Types

Old data type name	New data type name	Description
Byte	UInt8	unsigned 8-bit value
UChar	UInt8	unsigned 8-bit value
SByte	Int8	signed 8-bit value
Int	Int16	signed 16-bit value
SWord	Int16	signed 16-bit value
Short	Int16	signed 16-bit value
UShort	UInt16	unsigned 16-bit value
UInt	UInt16	unsigned 16-bit value
Word	UInt16	unsigned 16-bit value
Long	Int32	signed 32-bit value
SDWord	Int32	signed 32-bit value
ULong	UInt32	unsigned 32-bit value
DWord	UInt32	unsigned 32-bit value
Handle	MemHandle	a handle to a memory chunk
VoidHand	MemHandle	a handle to a memory chunk
Ptr	MemPtr	a pointer to memory
VoidPtr	MemPtr	A pointer to memory

D

Resource Tools

There are two tools provided with the Metrowerks CodeWarrior environment that you can use to work with resources:

- Use the Rez tool to compile a textual description of the resources for your application into a resource file.
- Use the DeRez tool to decompile a resource file into a text file.

Both of these tools are standard Apple Computer tools for working with MacOS application resources. Documentation for both the Rez and DeRez programs is found in the Apple book *Building and Managing Programs in MPW, 2nd Edition*. This book is available online at the following URL:

http://developer.apple.com/tools/mpw-tools/books.html

Glossary

Term	Definition
application error message	A message displayed when software running on the handheld device calls a system function such as `ErrDisplayFileLineMsg` or `SysFatalAlert.`
BigROM	The ROM code that initializes the hardware and contains all of the system code.
command request packet	A packet sent from a debugging host to a debugging target that requests a certain action.
command response packet	A packet sent from a debugging target to a debugging host in response to a command request packet, or in response to a state change.
console nub	A background thread on the handheld device that listens for commands on the serial or USB port. This thread provides the functionality required to perform database and heap management functions, and must be activated before Palm Debugger's console commands can be sent to the device.
debugger nub	A background thread on the handheld device that listens for commands on the serial or USB port. This thread provides the functionality required to support debugging of Palm OS applications, and must be activated before Palm Debugger's debugging commands can be sent to the device.
debugging host	The desktop computer that hosts the debugging program.
debugging target	The handheld device ROM or emulator running the executable that is being debugged.
gremlin	A series of user input events that test your your application's capabilities.

Term	Definition
gremlin horde	A group of gremlins that you can use to test specific capabilities.
heap scrambling	The process of moving items around in a heap to test if a program has correctly used handles for memory accesses. Direct memory pointers will no longer work after the heap has been scrambled, but handles do continue to work.
message packet	A packet sent from a debugging host to a debugging target
processor exception	An error condition that involves the CPU pushing the current program counter and processor state onto the stack, and then branching through a low-memory vector.
memory access exception	An error condition that involves access to a memory location the application is not supposed to access.
skin	A set of graphics that an application uses to creates its appearance. You can change the appearance of an application such as the Palm OS Emulator by changing its skin.
SmallROM	The bootstrap code on the handheld device. This is the code at the very front of the device ROM that can initialize the hardware and activate the debugger nub.

Index

D

E

F

Q

R

S

T

U

V

W