Palm OS® Emulator

Excerpt from Palm OS Programming Development Tools Guide

Document Number 3011-003
Print Date 2/01

CONTRIBUTORS

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

Palm OS Programming Development Tools Guide
Document Number 3011-003
February 20, 2001
For the latest version of this document, visit
http://www.palmos.com/dev/tech/docs/.

Palm, Inc.
5470 Great America Pkwy
Santa Clara, CA 95054
USA
www.palmos.com

Table of Contents

1

Using Palm OS® Emulator

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

This edition covers Palm OS Emulator 3.0a8.

Note: 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.

This chapter begins with overview information:

Next, the chapter addresses how you can use an Emulator session to test your application:

The remainder of the chapter covers additional concepts and reference information:

About Palm OS Emulator

Palm OS Emulator is a hardware emulator program for the Palm Powered 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 Palm OS Emulator on your desktop computer, 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 Powered handhelds. The difference is that Palm OS Emulator executes these instructions in software on your desktop computer.

Feature Overview

Palm OS Emulator displays an on-screen image that looks exactly like a Palm Powered handheld, as shown in Figure 1.1.

Figure 1.1 Palm OS Emulator display

You can select which type of Palm handheld device you want to emulate. You can also specify whether you want 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.

You can use the mouse on your desktop computer just as you use the stylus on a Palm Powered handheld. 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.

You can use Palm OS Emulator to perform some debugging of your applications, and you can use Emulator with Palm Debugger to perform extensive debugging of your applications. When you connect Emulator with Palm Debugger, you can debug in exactly the same manner as debugging with your application running on an actual hardware handheld device. For more information about Palm Debugger, see Chapter 4, "Using Palm Debugger."

Standard Device Features

Palm OS Emulator accurately emulates Palm Powered 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 (mouse)
- 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:

- use of 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

Prerequisites

Palm OS Emulator Runtime Requirements

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

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

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 Resource Pavilion, 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" on page 18.

When you download ROM images from the Palm Resource Pavilion, you can also obtain debug ROM images. Debug ROM

images contain additional error checking and reporting functions that can help you debug Palm OS applications.

For more information about testing and debugging applications with Palm OS Emulator, see "Testing Your Application" on page 43.

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.palmos.com/dev

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

Note: For the Unix version of Palm OS Emulator, the source code is provided rather than the executables listed in the table below.

Table 1.1 Files Included in the Palm OS Emulator Package

File name	Description
• `Emulator.exe` (Windows) • `Palm OS Emulator` (Macintosh)	Main Palm OS Emulator executable
• `Emulator_Profile.exe` (Windows) • `Palm OS Emulator - Profile` (Macintosh)	Palm OS Emulator with added profiling facilities

Table 1.1 Files Included in the Palm OS Emulator Package

File name	Description
`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 • `_Building.txt`, which describes how to build Emulator executables
• `ROM Transfer.prc` (Windows, Macintosh) • `ROM_Transfer.prc` (Unix)	Palm OS application used to transfer the ROM image from your handheld device to your desktop.
`HostControl.h`	C/C++ header file declaring functions that can be used to control Palm OS Emulator. For more information about the Host Control API, see Chapter 2, "Host Control API Reference."

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
3.0	Official release version 3.0
2.1d19	The 19th developmental release of version 2.1.
3.0a8	The 8th alpha release of version 3.0.

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" on page 61.

Loading ROM Images

Because Palm OS Emulator emulates the Palm Powered 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 Resource Pavilion
- transfer a ROM image from a handheld

Downloading a ROM Image Obtained From Palm

To download a debug ROM image from Palm, see:

http://www.palmos.com/dev

The ROM image files are found in the Resource Pavilion.

The Resource Pavilion is an area for developers who have registered as members of the Palm Alliance Program. You can find instructions for joining the Palm Alliance Program at the developer site.

Transferring a ROM Image From a Handheld

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

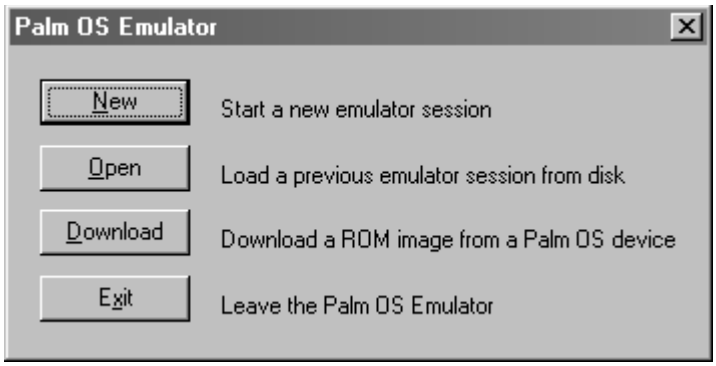
1. Install the Palm OS application named `ROM Transfer.prc` on your handheld device. You can use the Install program in the Palm Desktop organizer software and then synchronize with the handheld to install this program.
2. Place the handheld in the HotSync cradle that is connected to your desktop computer.
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 1.2. Click **Download** to begin the transfer of a ROM image from a handheld.

Figure 1.2 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 "Starting Palm OS Emulator" on page 28.

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 **Transfer ROM**.

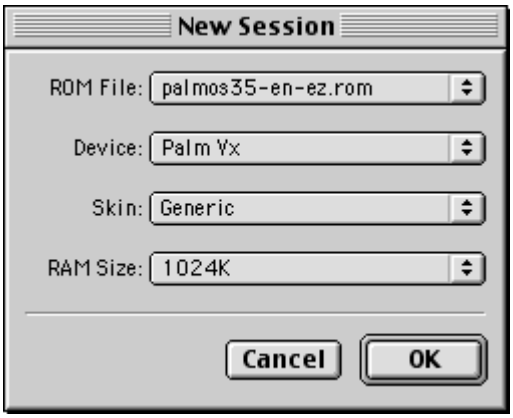
Palm OS Emulator opens a Transfer ROM dialog box that will guide you through the process.

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 1.3.

Figure 1.3 Running Palm OS Emulator for the First Time on a Macintosh System



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

Palm OS Emulator opens a Transfer ROM dialog box that will guide you through the process.

Transferring a ROM File On a Unix System

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

When running the program on a Unix system, Palm OS Emulator presents an empty window frame as shown in Figure 1.4 on page 22.

Figure 1.4 Running Palm OS Emulator for the First Time on a Unix System

Right-click (use mouse button two) on the window to display the Emulator popup menu. Click **Transfer ROM** to begin the transfer of a ROM image from a handheld.

Palm OS Emulator opens a Transfer ROM dialog box that will 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 **New** from the popup menu. Table 1.3 shows the first step in creating a new session for each transfer method.

Table 1.3 Initiating a New Session After Transferring a ROM Image

Method Used to Initiate ROM Transfer	New Session Method
Clicked **Download** initial dialog box in Windows	Click **New** in the dialog box.
Selected **Transfer ROM** in Windows	Select either **New** or **Close** from the File menu.
Selected **Transfer ROM** on a Macintosh	Select **New** from the File menu.
Selected **Transfer ROM** on Unix	Select **New** from the File menu.

After you initiate the session, Palm OS Emulator presents the new session dialog box, which is described in "Configuring a New Session" on page 31. The Windows version of this dialog box is shown in Figure 1.5.

Figure 1.5 New Session 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" on page 33.

Running Palm OS Emulator

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 on page 13.

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 1.4 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, as described in "Configuring a New Session" on page 31.

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

Table 1.4 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.
`-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.

Table 1.4 Palm OS Emulator Command Line Options

Option syntax	Parameter values	Description
`-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.
`-rom <fileName>`	Any valid ROM file name	The name of the ROM file to use.

`-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.

Table 1.4 Palm OS Emulator Command Line Options

Option syntax	Parameter values	Description
`-device <type>`	One of the following device type values: `Pilot`, `Pilot1000`, `Pilot5000`, `PalmPilot`, `PalmPilotPersonal`, `PalmPilotProfessional`, `PalmIII`, `PalmIIIc`, `PalmIIIe`, `PalmIIIx`, `PalmV`, `PalmVx`, `PalmVII`, `PalmVIIEZ`, `PalmVIIx`, `m100`, `Symbol1700`, `TRGpro`, `Visor`	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` files 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.
`-quit_on_exit`	None	If the `-run_app` option was specified, this option indicates that Palm OS Emulator should quit after that application terminates.

Table 1.4 Palm OS Emulator Command Line Options

Option syntax	Parameter values	Description
`-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>`	The name of a skin. The skin names are defined by the device-specific `.skin` files. For most devices, these skin names are available:
	`Generic`
	`Standard-English`
	`Standard-Japanese`
		The skin types to emulate during the session.

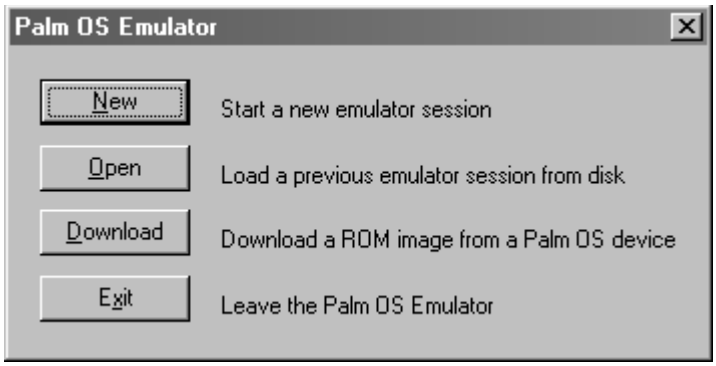
Starting Palm OS Emulator

The most common scenario for starting Palm OS Emulator is without any command line parameters. In this case, Emulator restarts with saved information from the previous session.

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 1.6.

Figure 1.6 Palm OS Emulator Startup Dialog Box



NOTE: The dialog box shown in Figure 1.6 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 1.3 on page 21 is displayed instead.

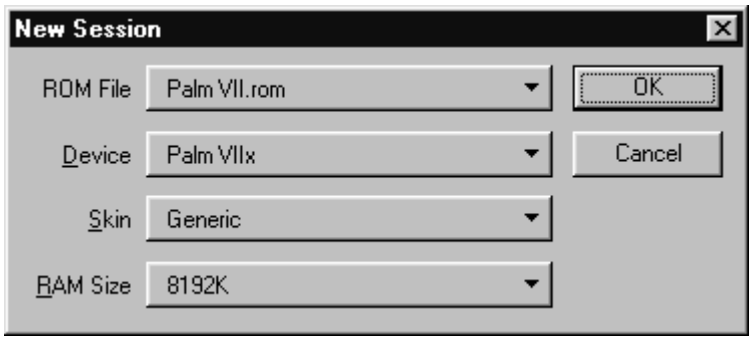
If you are using a Unix system, Palm OS Emulator does not provide an automatic startup sequence; instead, it presents you with a window shown in Figure 1.4 on page 22, and you must right-click in that window to display the new session menu.

2. If you are using Windows or Unix with command line options specified:
   - 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.
   - 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.
   - 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 1.7.

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 1.7 New Session Dialog Box



3. 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.
4. 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.

NOTE: When it starts up, Palm OS Emulator looks for the most recently saved PSF file:
- On Windows and Unix, Emulator uses the full path name of that file.
- On Macintosh, Emulator uses aliases to locate the file.

If Emulator cannot find that file, it looks for the file name in the directory in which the Palm OS Emulator executable is located.

Using Emulation Sessions

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

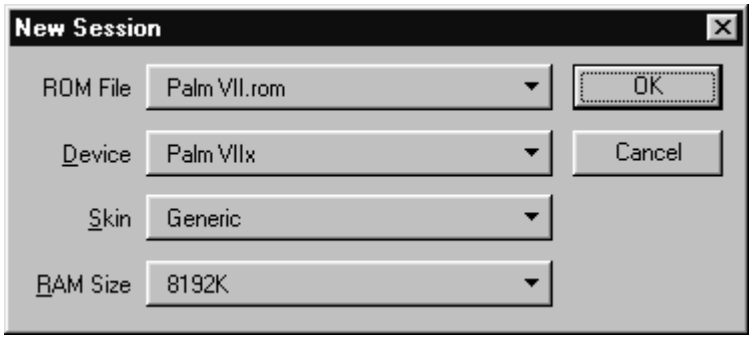
You can start new emulation sessions during a single run of Palm OS Emulator. You can also save the current state of a session and restore it in a later session. This session describes these features of Palm OS Emulator.

Configuring a New Session

You can start a new session in Palm OS Emulator by choosing **New** 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 1.8 shows the New Session dialog box, which Palm OS Emulator displays when you choose **New** from the menu.

Figure 1.8 Configuring a New Session



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

- Select the ROM file on your desktop computer that you want to use for the session. You can click on the arrow and select

Other... to navigate to the file. For more information about ROM files, see "Loading ROM Images" on page 18.

- Select the Palm handheld device that you want to emulate in the session. Only those devices that apply to the selected ROM will be shown in the list. The list may include the following choices:

-Pilot	-Pilot 1000	-Pilot 5000
-PalmPilot	-PalmPilotPersonal	-PalmPilotProfessional
-Palm III	-Palm IIIc	-Palm IIIe
-Palm IIIx	-Palm V	-Palm Vx
-Palm VII	-Palm VIIEZ	-Palm VIIx
-m100	-Symbol1700	-TRGpro
-Visor		

- Select the skin that you want displayed on the emulation screen.

  Note that the skin is simply a graphic; it does not change the ROM or the device being emulated. The skin simply changes the appearance of the Emulator window.

  The skin choices available are dependent on the device selection. When you select a device, Emulator reads through the available SKIN files for the skin names that support the selected device.

  Alternative skins, such as the Japanese skin, are only available for certain device types. The **Generic** choice is always available, even when alternatives are not available. For additional information, see the section "Changing the Emulator's Appearance" on page 34.

- 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 (1024K) is most often the right amount of RAM to emulate. Using 1 MB of RAM tells you if your application will work properly across the majority of hardware devices available.

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

The Difference Between the New Menu Item and the Open Menu Item

Both **New** and **Open** can be used to initiate an emulator session. However, the **Open** menu is used to open an existing session file (PSF file) that has been saved from a previous emulator session. The **Open** menu does not allow you to change the ROM file or device being emulated.

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, 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`, `PDB`, 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**, which saves the contents of the emulated Palm handheld device screen.

Figure 1.9 A Palm OS Emulator Screen Shot

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

Changing the Emulator's Appearance

You can change the appearance of Palm OS Emulator by choosing **Skins** from the **Settings** submenu. This displays the Skins dialog box, which is shown in Figure 1.10.

Figure 1.10 Changing Palm OS Emulator Appearance

The Skins dialog box lists the skins that are available for the device that is being emulated. This means that you cannot use a Palm V skin for a Palm III device, for example.

Emulator comes with a built-in **Generic** skin. You can download additional skins from:

http://www.palmos.com/dev/tech/tools/emulator/

When you download the package of additional skins, you place the skins in a skins directory. When you select a device during session configuration, Emulator reads through the skins directory and finds all of the skins that can be used with the selected device. The supported skins are displayed in the Skins dialog box.

Note that the skin is simply a graphic. Selecting a skin changes the appearance of the Emulator window, but it does not change the ROM or the device being emulated.

Other Options on the Skins Dialog Box

In addition to selecting a skin, use the Skins dialog box to change these appearance options:

- Select or deselect **Double scale** to display the emulated device in double size or actual size on your monitor.

- Select or deselect **White Background** to display the emulated device LCD background color in white or green on your monitor.

Modifying the 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

Use the Properties dialog box to modify characteristics of your Palm OS Emulator sessions. To display this dialog box, choose **Properties** on Windows or **Preferences** on Macintosh or Unix. The Properties dialog box is shown in Figure 1.11.

Figure 1.11 Changing Palm OS Emulator Properties

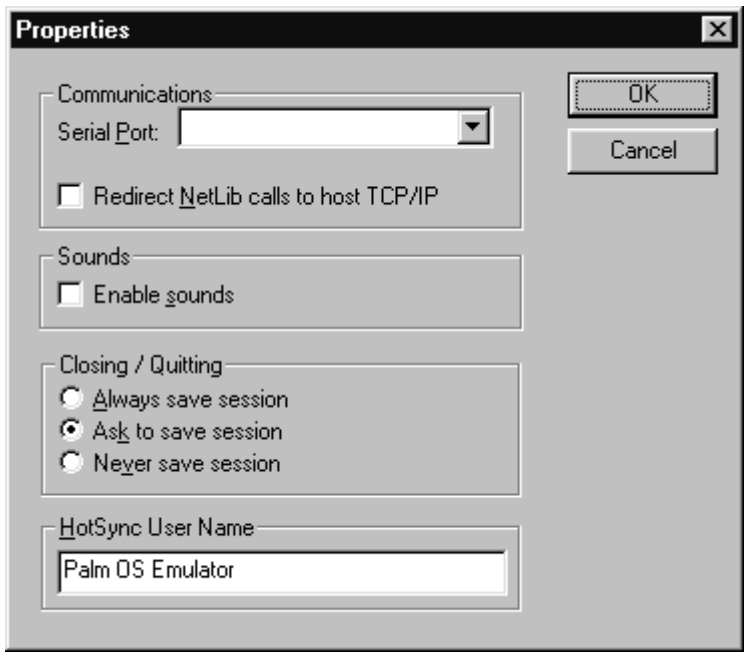


Table 1.5 describes the options available in the properties dialog box.

Table 1.5 Palm OS Emulator properties

Option	Description
Serial Port	Specifies which serial port 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.
Enable sounds	Specifies whether Palm OS Emulator should enable emulation of device sounds.
Session saving	Selects what action 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 1.6.

Table 1.6 Palm OS Emulator Preference File Locations

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

Installing Applications

Use **Install** to load applications or databases directly into the current Palm OS Emulator session.

- For Windows and Unix, right-click on the Palm OS Emulator screen display and choose **Install Application/Database**
- On a Macintosh system, select **Install Application/Database** from the File menu

Install displays an open file dialog box in which you can choose the application (`PRC`), database (`PDB`), or Palm Query Application (`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.

IMPORTANT: 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. Use **Install** while an application is running in the emulated session.

Serial Communications and Palm OS Emulator

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 **Properties** (on Macintosh and Unix computers, this is **Preferences**), as described in Palm OS Emulator Properties.

When emulated software accesses the processor 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 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. This method is described in the "Emulating Network Hotsync with Palm OS Emulator on Windows" section below.
- Alternatively, you can use a null-modem cable to connect two serial ports together and perform a HotSync operation. This method is described in "Emulating HotSync with a Null Modem Cable" section below.

Emulating Network Hotsync with Palm OS Emulator on Windows

You do not need to be connected to a network to emulate Network HotSync with Palm OS Emulator. This method can be used with Emulator and a single Windows computer. However, other configurations are possible.

In general, you need these two:

- a Windows computer running HotSync Manager
- a computer running Emulator that can access the computer running HotSync Manager.

The computer running Emulator can be the same Windows computer that is running HotSync Manager, or it can be a second computer (either Windows, Macintosh, or Unix). If you are using a single Windows computer, you don't need to be connected to a network. However, if you are using a second computer, you will need the actual IP address of the Windows computer running HotSync Manager for step 4 below.

Here is the complete process for emulating Network HotSync:

1. Ensure that you have the Network HotSync application on your emulated device:
   - If you are emulating a Palm III or m100 device, you must first download and install the Network HotSync

application on the emulated device. You can get the Network HotSync files from:

http://www.palm.com/support/downloads/netsync.html

- If you are emulating a device running Palm OS version 3.1 or later, then you may already have the Network HotSync application installed on the emulated device.

2. Configure HotSync on your Windows computer:
   - Right-click (use mouse button two) on the HotSync icon in the system tray.
   - In the pop-up menu, select **Network** to enable Network HotSync. (A checkmark will appear next to the **Network** menu item if it is already enabled.)
3. Configure Palm OS Emulator to Redirect NetLib Calls to TCP/IP:
   - Right-click (use mouse button two) on Emulator.
   - In the pop-up menu, select **Settings>Properties...**
   - In the Properties dialog box, click the **Redirect NetLib Calls to TCP/IP** checkbox. Click **OK** to save the changed properties.
4. Configure HotSync on the emulated device:
   - From the device's application launcher, tap the HotSync application to open it.
   - Tap **Menu** to display the HotSync application's menu.
   - Select **Options>Modem Sync Prefs...**
   - In the Modem Sync Preferences dialog box, tap the **Network** button. Tap the **OK** button to save the changed preferences.
   - Tap **Menu** to display the HotSync application's menu again.
   - Select **Options>LANSync Prefs...**
   - In the LANSync Preferences dialog box, tap the **LANSync** button. Tap the **OK** button to save the changed preferences.

– Tap **Menu** to display the HotSync application's menu again.

– Select **Options>Primary PC Setup...**

– In the Primary PC Setup dialog box, enter the **Primary PC Address** (the middle entry field):

- If you are running Emulator and HotSync manager on the same Windows computer, enter `127.0.0.1`

- If you are running Emulator on a second computer, then enter the actual IP address of the Windows computer running Network HotSync.

Tap the **OK** button to save the changed preferences.

– In the HotSync application, tap **Modem**. Next, tap the **Select Service** button under the Modem Sync icon.

– In the Preferences dialog box, tap the **Tap to enter phone** field. In the Phone Setup dialog box, enter `00` in the **Phone #** entry field. Then tap the **OK** button. Then tap the **Done** button.

– To start the HotSync, tap the HotSync icon in the center of the HotSync dialog box.

Emulating HotSync with a Null Modem Cable

You can emulate HotSync by connecting the serial port that the HotSync application uses to communicate with the handheld device to another serial port that Palm OS Emulator uses. You connect these ports together with a null modem cable, such as a LapLink cable.

For example, if your HotSync application uses the COM1 port, follow these steps:

1. Select **Properties** (**Preferences** on a Macintosh or Unix) and specify the COM2 port for Palm OS Emulator.
2. Connect COM1 and COM2 together with a null modem cable.
3. Select **HotSync** from the Palm OS Emulator menu.

The HotSync application synchronizes with 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 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 re-enable 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 Palm OS Emulator.
5. Press F10 to display the menu, then H to begin the HotSync operation.
6. After the HotSync operation completes, re-enable your mouse.
7. Restart Windows again.

TIP: When you first perform a HotSync operation with 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.

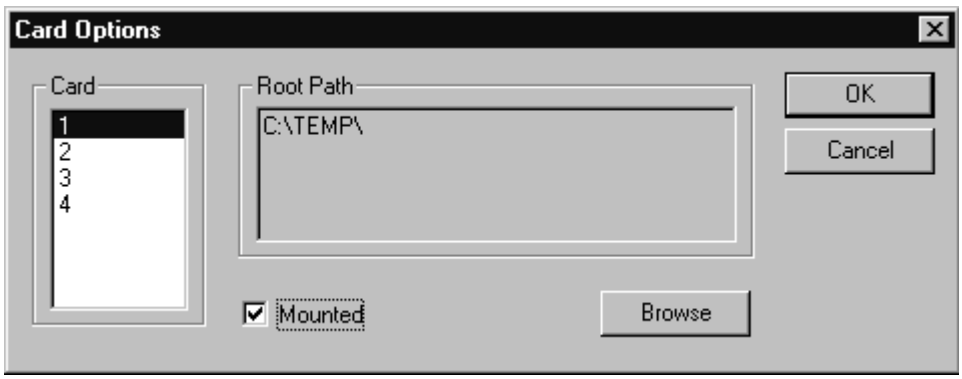
Emulating Expansion Cards

Palm OS 4.0 includes the Expansion Manager, which manages plug-in memory cards, and the Virtual File System manager, which supports the management of files on memory cards.

Palm OS Emulator can emulate these cards, which the Expansion Manager will recognize and mount in the same way it would mount an actual hardware expansion card. The Virtual File System Manager will then read from and write to the host operating system using the mount information associated with the emulated card. The host operating manipulation is performed using the many file-related host control functions available. (See "Host Control API Reference" on page 81 for more information on the host control API.)

To specify mount information for card emulation, use the Card Options dialog box shown in Figure 1.12 on page 43.

Figure 1.12 Palm OS Emulator Card Options Dialog Box



The Card Options dialog box supports the mounting of up to four emulated cards. For each card, you can specify a directory in the host file system that will serve as the root for the card as managed by the Virtual File System Manager. You can also specify whether a particular card is actually mounted.

You can change the card options settings while an emulation session is running. Changes regarding whether a card is mounted or not take place immediately; the Palm OS is notified that the card has been added or removed. Changes regarding the root path take effect only when the card is mounted.

Testing Your Application

This section describes how to use Palm OS Emulator to test and/or debug an application.

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

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 OS's programming guidelines, the Emulator displays a dialog box that explains what is happening.

The debugging options dialog box, which is shown in Figure 1.13, lets you enable or disable the monitoring activities applied to your application. Use **Debug Options** to display this dialog box.

Figure 1.13 Palm OS Emulator Debug Options Dialog Box

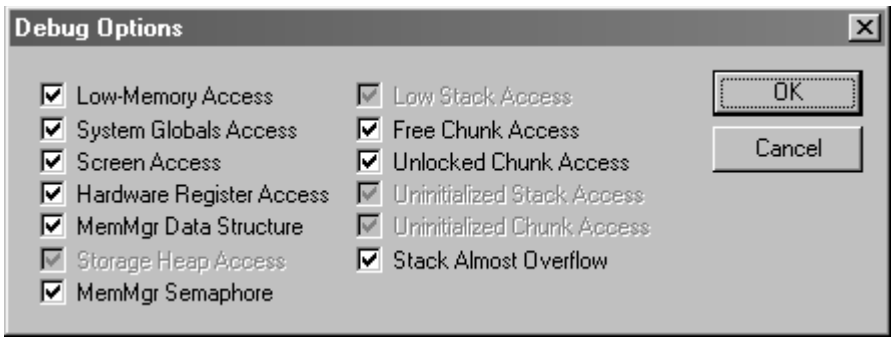


Table 1.7 describes each of the debugging options.

Table 1.7 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 memory location in the range from `0x0100` to the end of the trap dispatch table.
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.

Table 1.7 Emulator Debugging Options *(continued)*

Option	Description
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.
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.

Table 1.7 Emulator Debugging Options *(continued)*

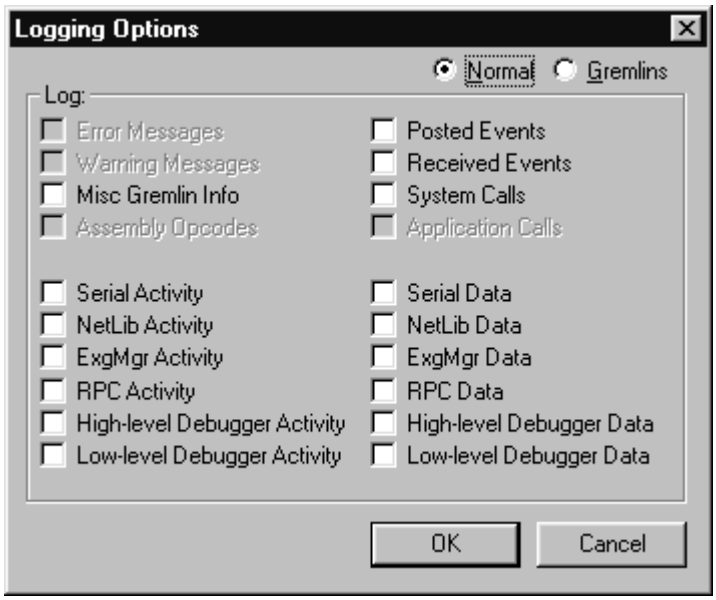
Option	Description
Uninitialized Chunk Access	Monitors read access to uninitialized portions of memory chunks that have been allocated by the `MemChunkNew`, `MemPtrNew`, and `MemHandleNew` functions. You can use this option to detect read accesses to uninitialized portions of dynamically allocated memory 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 Emulator 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

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 1.14. Use **Logging Options** to display this dialog box.

Figure 1.14 Palm OS Emulator Logging Options Dialog Box



The logging options dialog box features radio buttons to indicate logging during normal operations (**Normal**), and logging while a Gremlin is running (**Gremlins**). Both offer the same options, which are described in Table 1.8

Table 1.8 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.
Assembly Opcodes	Logs assembly-level trace information, including registers, the program counter, opcodes, and related information. This option is not yet implemented.

Table 1.8 Emulator Logging Options *(continued)*

Option	Description
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 1.8 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	Logs calls to `ExgMgr` functions.
ExgMgr Data	Logs data sent and received via `ExgMgr` calls.
RPC Activity	Logs remote procedure calls.
RPC Data	Logs data sent and received via remote procedure calls.
High-level Debugger Activity	Logs messages received back from an external debugger, and the messages sent back to the debugger.

Table 1.8 Emulator Logging Options *(continued)*

Option	Description
High-level Debugger Data	Logs details of the messages sent to and received from an external debugger.
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 to Automate Testing

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

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 **Stop** or **Step** from the Emulator menu or from the Gremlin Status dialog box.

If a Gremlin generates an error, it is halted and 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.

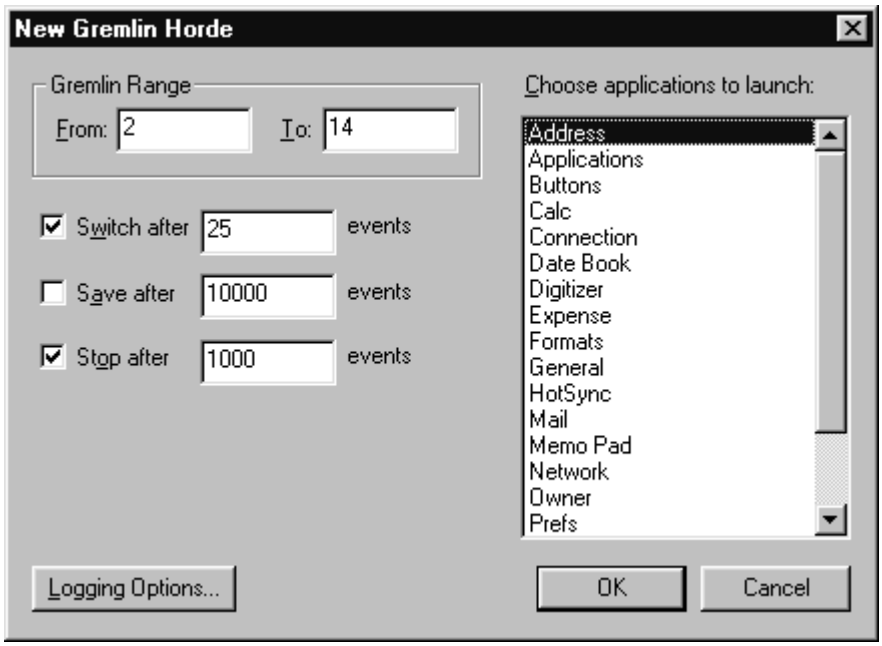
- Reloads the original session state.
- Starts the second Gremlin and run it until it posts that number of events, at which time its state is saved to another file, and the Gremlin is suspended.
- Runs each Gremlin in the horde, until each has been suspended or terminated:
  - A Gremlin is terminated 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 **New Gremlin** to start a Gremlin. The new Gremlin dialog box displays, as shown in Figure 1.15. Use this dialog box to specify the characteristics of the Gremlin horde that you want to run.

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

Figure 1.15 The Gremlin Horde Dialog Box



When Palm OS Emulator runs the example shown in Figure 1.15, 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.

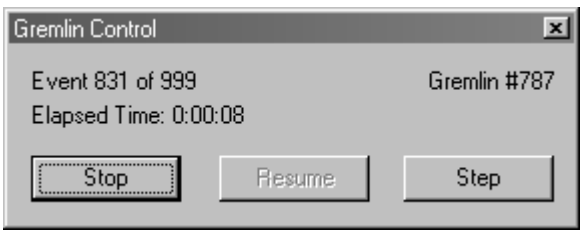
3. Similarly, each Gremlin (#4 through #14) runs and receives twenty-five events, after which it is suspended.
4. The Emulator loops back to Gremlin #2 and runs it, sending it twenty-five events before again suspending it.
5. Gremlin #3 runs again, receives twenty-five events, and suspends.
6. This looping 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 1.16. You can use the commands in this dialog box to stop, resume, and single-step a Gremlin. You can also use the **Gremlins** menu to perform these actions.

Figure 1.16 The Gremlin Status Dialog Box



Gremlin Snapshots

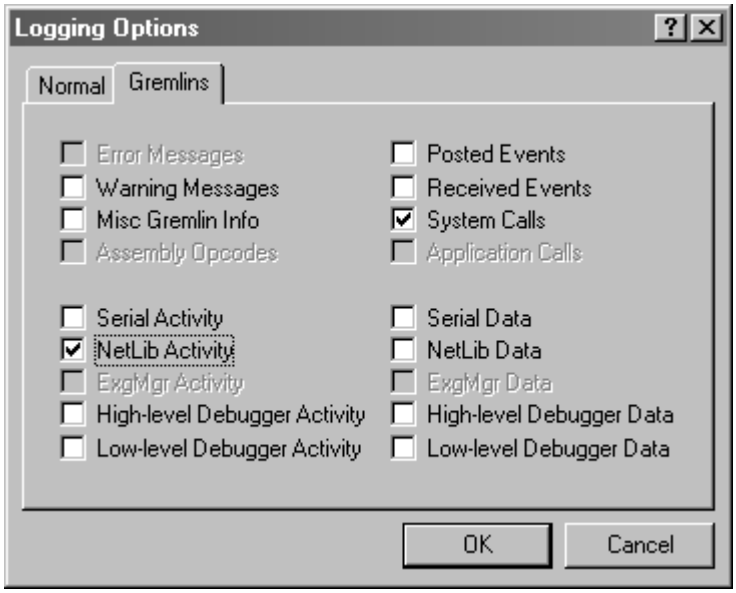
When you start a new Gremlin horde, you can specify that you want Palm OS Emulator to take a snapshot on a regular basis. You specify a frequency value, as shown in Figure 1.15 on page 54, and the

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 lets you specify separate logging options to use while Gremlins are running. Figure 1.17 shows the Gremlin logging options dialog box. Each of the options is described in "Logging Options" on page 47.

Figure 1.17 Gremlin logging options



Setting Breakpoints

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

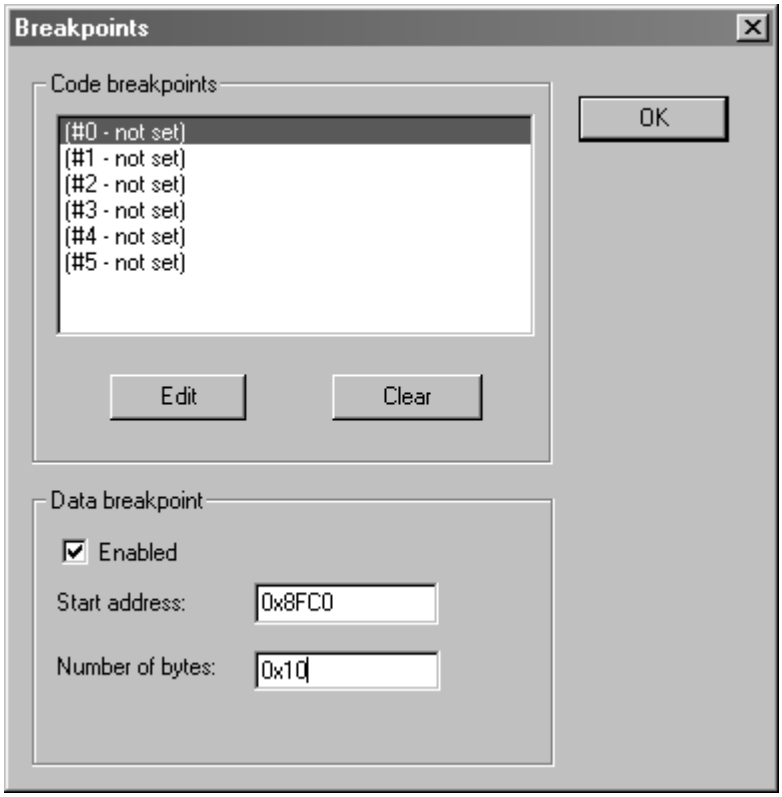
- If you are running the Emulator connected to a debugger, the Emulator sends a message to the debugger, informing it that the breakpoint was hit. The debugger then handles that command as it sees fit.

- 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 **Breakpoints** from the **Settings** menu. The Breakpoints dialog box is displayed, as shown in Figure 1.18.

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

Figure 1.18 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 in either hexadecimal (`0x`) or decimal.

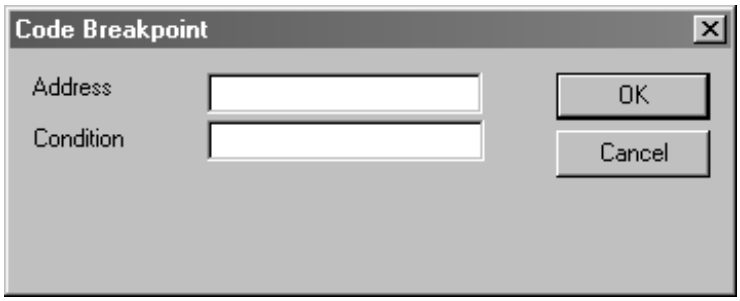
Setting Conditional Breakpoints

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 **Edit**. This displays the Code Breakpoint dialog box, which is shown in Figure 1.19.

Figure 1.19 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.

IMPORTANT: 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 Emulator with Palm Debugger

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

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

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

Connecting Emulator with the GDB Debugger

You can use the gdb debugger with Palm OS Emulator to debug your applications. To use the gdb debugger with an emulator session, follow these steps:

1. When you build your application, both compile and link with the `-g` option (that is, using "`gcc -g ...`"). When you compile using the `-g` option, the compiler generates the necessary symbol information. When you link using the `-g` option, the linker forces the inclusion of a debug runtime routine that installs a breakpoint in `PilotMain`.
2. Start Palm OS Emulator, and install your application in the emulator session.
3. Start the gdb debugger, loading your application's symbol table (for example, using "`gdb myApp`"). Note that the file to be loaded is the `myApp` file created by the gcc linker, not the `myApp.prc` created by buildprc.

4. In the gdb debugger, enter "`target pilot localhost:2000`". The gdb debugger will respond with a message indicating that remote debugging is starting.
5. Start your application on Palm OS Emulator.
6. Wait for the gdb debugger to see the initial breakpoint and prompt you, then start debugging.

Connecting the Emulator With External Debuggers

Palm OS Emulator can communicate with external debuggers using the methods shown in Figure 1.16.

Table 1.9 Palm OS Emulator Connections

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

NOTE: Currently, Palm Debugger uses TCP only when running on Windows. The CodeWarrior plug-in uses TCP if you select `Use sockets` 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 Palm Debugger 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 communications, which you do by setting the `DebuggerSocketPort` preference to `0`.

Profiling Your Code

One of the features of 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.

NOTE: In order to use the profiling features, you must be using a version of Palm OS Emulator with profiling enabled.

On Windows and Macintosh, this means that you must be using the executable with "`profile`" in its name. See Table 1.1 on page 16 for more information.

On Unix, this means that you must build the executable with the configure switch "`--enable-profile`". (See the `_Building.txt` file mentioned in Table 1.1.)

You can start a profiling session by choosing **Profiling Start**. While profiling is active, 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 **Profiling Stop**.

You can then save the profiling information to a file by selecting **Profiling Dump**. 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.

Error Handling Concepts

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

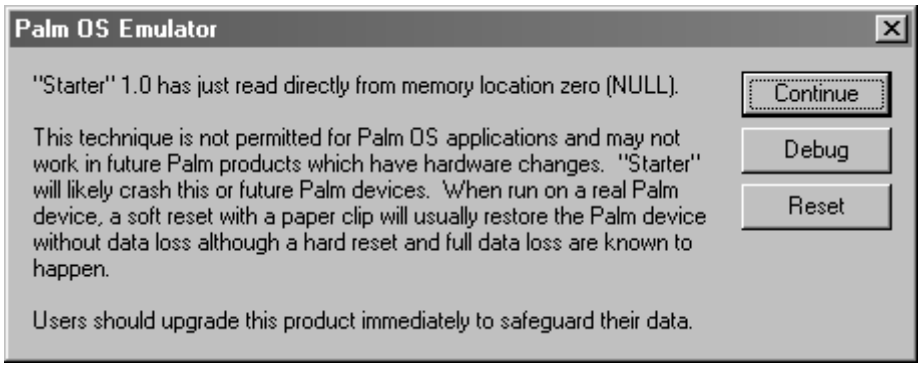
- which conditions are detected
- what the Emulator does upon detecting an error condition
- the message displayed for each error condition
- the options available to the user when an error condition occurs

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 click **Debug** 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 1.20.

Figure 1.20 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

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`.

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 1.10 shows 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 1.10 Palm OS Emulator error messages

Error type	Description	Message example
Hardware register access	The application or system software has accessed a processor 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.`
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.`

Table 1.10 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
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" .`
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.`

Table 1.10 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
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.`
Memory Manager semaphore acquisition 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.`

Table 1.10 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example

Invalid heap	Heap corruption detected during a regular heap check. 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`

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.`

Table 1.10 Palm OS Emulator error messages *(continued)*

Error type	Description	Message example
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 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.`
SysFatalAlert	The application or system software has called the `SysFatalAlert` function. 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 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.`

Advanced Topics

Creating Demonstration Versions of Palm OS Emulator

If you are running Palm OS Emulator on Windows NT, you can 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.

To save a demonstration version of the Emulator session, you can right-click on the Palm OS Emulator display (the Palm device image) and select **Save Bound Emulator...**

Sending Commands to Palm OS Emulator

You can use RPC packets to send commands to Palm OS Emulator. You can invoke any function in the Palm OS dispatch table, including the Host Control functions, which are described in Chapter 2, "Host Control API Reference."

The RPC packets use the same format as do packets that are sent to the debugger interface, which is described in Chapter 3, "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 Chapter 3, "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 Chapter 3, "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 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.

User Interface Summary

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

Palm OS Emulator Display

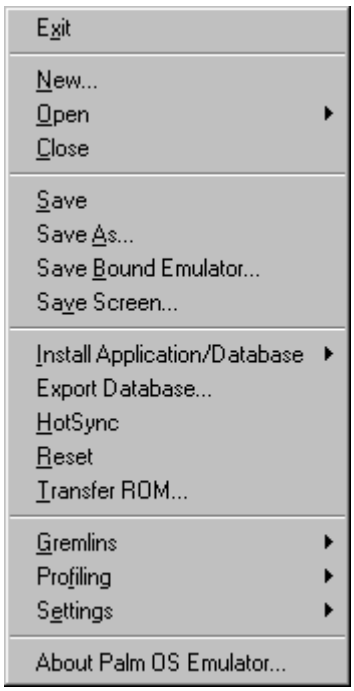
The Palm OS Emulator display looks very much like a real Palm Powered handheld device. You can use your mouse to perform actions that you perform with the stylus on handheld devices, and you can use the menus to access Palm OS Emulator functionality.

Using the Menus

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

- If you are using Windows, 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 1.21.

Figure 1.21 The Windows Version of the Palm OS Emulator Menu



- If you are using a Macintosh, select menu items from the menu bar. The Macintosh menu presents the same items in four different menus, as described in Table 1.11. The Macintosh version is only slightly different:
  - The Macintosh version of Palm OS Emulator uses **Preferences** instead of **Properties** to access the option-setting dialog box.
  - The Macintosh version of the Emulator features **Undo**, **Cut**, **Copy**, **Paste**, and **Clear**, which are not available in the Windows version.
  - The Macintosh version of the Emulator uses **Quit** instead of **Exit**.
  - The Macintosh version does not feature **Breakpoints**.

Table 1.11 Palm OS Emulator Macintosh Menus

Menu	Commands
File	**New** **Open** **Close** --- **Save** **Save As** **Save Bound Emulator** **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 items as are included with the Windows version, except that **Breakpoints** is not available. The Unix version of the menu pops up like the Windows version, and uses a different hierarchy, but presents the same items.

Table 1.12 provides a brief description of the Palm OS Emulator menu items, listed in alphabetical order.

Table 1.12 The Palm OS Emulator Menu Items

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 menu item 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 to Automate Testing" on page 51.
HotSync	Lets you synchronize the emulator session environment with the desktop computer. See "Using the HotSync Application With Palm OS Emulator" on page 39 for more information about the cabling requirements and other considerations for this menu item.

Table 1.12 The Palm OS Emulator Menu Items *(continued)*

Command	Description
Install App/DB	Lets you 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" on page 37.
Export Database...	Exports a database to your desktop computer as a `PDB` or `PQA` file, or exports an application to your desktop computer as a `PRC` file.
New	Displays the New Session dialog box. The New Session dialog box lets you select the session's ROM file, device, skin, and RAM size.
Open	Displays the open file dialog box for opening a saved emulator session file. Note that the **Open** menu is for opening saved session files (PSF files), not for opening ROM files. To change the ROM file for your emulator session, you need to use the **New** menu.
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" on page 61.
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.
Save Bound Emulator	Saves the current emulator session as an executable, which can be used for demonstration purposes.

Table 1.12 The Palm OS Emulator Menu Items *(continued)*

Command	Description
Save Screen	Saves the current screen image as a bitmap file. **TIP: Save Screen** 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" on page 36.
Settings: Logging	Presents the logging options dialog box, as described in "Logging Options" on page 47.
Settings: Debug	Presents the debug options dialog box, as described in "Debug Options" on page 44.
Settings: Skins	Presents the skins dialog box, as described in "Changing the Emulator's Appearance" on page 34.
Settings: Card Options	Presents the card options dialog box, as described in "Emulating Expansion Cards" on page 42.
Settings: Breakpoints	Presents the breakpoints dialog box, as described in "Setting Breakpoints" on page 56.
Transfer ROM	Lets you 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" on page 19.

Using the Hardware Buttons

Palm OS Emulator emulates each of the hardware buttons on Palm Powered 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 lets you activate the hardware buttons with keyboard equivalents, as shown in Table 1.13.

Table 1.13 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 lets you 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 lets you 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 1.14, are the same control keys that you can use with the Palm OS Simulator program.

Table 1.14 Palm OS Emulator Control Keys

Control key combination	Description
CONTROL+A	Displays the menu
CONTROL+B	Low battery warning
CONTROL+C	Command character
CONTROL+D	Confirmation character

Table 1.14 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+P	Jumps to the previous field
CONTROL+S	Automatic off character
CONTROL+T	Sets or unsets hard contrasts
CONTROL+U	Turns backlighting on or off

Getting Help With Palm OS Emulator

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

Palm provides a forum (emulator-forum@news.palmos.com) for questions and comments about Palm OS Emulator. To subscribe to the forum, see:

http://www.palmos.com/dev/tech/support/forums/

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

http://www.palmos.com/dev/

Note: The source code for Palm OS Emulator is available at:

http://www.palmos.com/dev/tech/tools/emulator/

You can create your own emulator by modifying this source code.

For more information on the protocol used in Palm OS Emulator to send requests to and receive responses from a debugging target, see Chapter 3, "Debugger Protocol Reference."

2

Host Control API Reference

This chapter describes the host control API. The following topics are covered in this chapter:

- "About the Host Control API" - Conceptual information about the host control API
- "Constants" on page 82 - A list of the constants that can be used with the host control functions
- "Data Types" on page 87 - A list of the data types that can be used with the host control functions
- "Functions" on page 94 - A list of all host control functions, sorted alphabetically
- "Reference Summary" on page 136 - A summary of all host control functions, sorted by category

About the Host Control API

You can use the host control API 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.

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.

IMPORTANT: This chapter describes the version of the host control API that shipped with Palm OS Emulator 3.0a8. If you are using a different version, the features in your version might be different than the features described here.

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 `HostErrType` value.

```
enum
{
hostErrNone = 0,
hostErrBase = hostErrorClass,
hostErrUnknownGestaltSelector,
hostErrDiskError,
hostErrOutOfMemory,
hostErrMemReadOutOfRange,
hostErrMemWriteOutOfRange,
hostErrMemInvalidPtr,
hostErrInvalidParameter,
hostErrTimeout,
hostErrInvalidDeviceType,
hostErrInvalidRAMSize,
hostErrFileNotFound,
hostErrRPCCall,
hostErrSessionRunning,
hostErrSessionNotRunning,
hostErrNoSignalWaiters,
hostErrSessionNotPaused,
hostErrPermissions,
hostErrFileNameTooLong,
hostErrNotADirectory,
```

```
  hostErrTooManyFiles,
  hostErrFileTooBig,
  hostErrReadOnlyFS,
  hostErrIsDirectory,
  hostErrExists,
  hostErrOpNotAvailable,
  hostErrDirNotEmpty,
  hostErrDiskFull,
  hostErrUnknownError
  };
```

`hostErrNone` No error.

`hostErrBase` A class error occurred.

`hostErrUnknownGestaltSelector`
The specified Gestalt selector value is not valid.

`hostErrDiskError`
A disk error occurred. The standard C library error code `EIO` is mapped to this error constant.

`hostErrOutOfMemory`
There is not enough memory to complete the request. The standard C library error code `ENOMEM` is mapped to this error constant.

`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. The standard C library error codes `EBADF`, `EFAULT` and `EINVAL` are mapped to this error constant.

`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. The standard C library error code `ENOENT` is mapped to this error constant.

`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`.

`hostErrPermissions`
: The standard C library error code `EACCES` and `EPERM` are mapped to this error constant.

`hostErrFileNameTooLong`
: The standard C library error code `ENAMETOOLONG` is mapped to this error constant.

`hostErrNotADirectory`
: The standard C library error code `ENOTDIR` is mapped to this error constant.

`hostErrTooManyFiles`
: The standard C library error code `EMFILE` and `ENFILE` are mapped to this error constant.

`hostErrFileTooBig`
: The standard C library error code `EFBIG` is mapped to this error constant.

`hostErrReadOnlyFS`
: The standard C library error code `EROFS` is mapped to this error constant.

`hostErrIsDirectory`
: The standard C library error code `EISDIR` is mapped to this error constant.

`hostErrExists`
: The standard C library error code `EEXIST` is mapped to this error constant.

`hostErrOpNotAvailable`
: The standard C library error codes `ENOSYS` and `ENODEV` are mapped to this error constant.

`hostErrDirNotEmpty`
: The standard C library error code `ENOTEMPTY` is mapped to this error constant.

`hostErrDiskFull`
: The standard C library error code `ENOSPC` is mapped to this error constant.

`hostErrUnknownError`
: The standard C library error code values that are not mapped to any of the above error constants are mapped to this error constant.

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`.

For a complete list of the constants available, see the `HostControl.h` header file.

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 Powered 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
};
```

`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.

Data Types

This section describes the data types that you use with the host control API.

HostBoolType

The host control API defines `HostBoolType` for use as a Boolean value.

```
typedef long HostBoolType;
```

New HostClockType

The host control API defines `HostClockType` as a platform-independent representation of the standard C library `clock_t` type.

```
typedef long HostClockType;
```

New HostDirEntType

The host control API defines `HostDirEntType` as a return value for the `HostReadDir` function. The contents are platform-specific, usually a simple null-terminated file name.

```
struct HostDirEntType
{
  char    d_name[HOST_NAME_MAX + 1];
};

typedef struct HostDirEntType HostDirEntType;
```

New HostDIRType

The host control API defines `HostDIRType` for use in directory-related functions. It is returned by HostOpenDir and used by HostReadDir and HostCloseDir. It represents an open directory whose contents can be read.

```
struct HostDIRType
{
  long    _field;
```

```
};
typedef struct HostDIRType HostDIRType;
```

HostFILEType

The host control API defines HostFILEType for the standard C library functions that take `FILE*` parameters. It is returned by HostFOpen and used by other host control functions. It represents an open file whose contents can be manipulated.

```
typedef struct HostFILEType
{
  long _field;
} HostFILEType;
```

HostGremlinInfoType

The host control API defines the `HostGremlinInfoType` structure type to store information about a horde of gremlins.

```
typedef struct HostGremlinInfoType
{
  long  fFirstGremlin;
  long  fLastGremlin;
  long  fSaveFrequency;
  long  fSwitchDepth;
  long  fMaxDepth;
  char  fAppNames[200];
};

typedef struct HostGremlinInfoType
HostGremlinInfoType;
```

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.

HostIDType

The host control API defines `HostIDType` for use as an identifier value.

```
typedef long HostIDType;
```

HostPlatformType

The host control API defines `HostPlatformType` for use as a platform identifier value.

```
typedef long HostPlatformType;
```

HostSignalType

The host control API defines `HostSignalType` for use in signal functions.

```
typedef long HostSignalType;
```

New HostSizeType

The host control API defines `HostSizeType` as a platform-independent version of the standard C library `size_t` type.

```
typedef long HostSizeType;
```

New HostStatType

The host control API defines `HostStatType` for status information about files.

```
struct HostStatType
{
        unsigned long   st_dev_;
        unsigned long   st_ino_;
        unsigned long   st_mode_;
        unsigned long   st_nlink_;
        unsigned long   st_uid_;
        unsigned long   st_gid_;
        unsigned long   st_rdev_;
        HostTimeType    st_atime_;
        HostTimeType    st_mtime_;
        HostTimeType    st_ctime_;
        unsigned long   st_size_;
        unsigned long   st_blksize_;
        unsigned long   st_blocks_;
        unsigned long   st_flags_;
};
typedef struct HostStatType HostStatType;
```

HostStatType Fields

`st_dev_`	Drive number of the disk containing the file (the same as `st_rdev_`) .
`st_ino_`	Number of the information node for the file (Unix-specific information).
`st_mode_`	Bit mask for file-mode information. The `_S_IFDIR` bit indicates if this is a directory; the `_S_IFREG` bit indicates an ordinary file or device. User read/write bits are set according to the file's permission mode; user execute bits are set according to the filename extension.
`st_nlink_`	Always returns 1 on non-NTFS file systems.

`st_uid_`	Numeric identifier of the user who owns the file (Unix-specific information).
`st_gid_`	Numeric identifier of the group who owns the file (Unix-specific information).
`st_rdev_`	Drive number of the disk containing the file (the same as `st_dev_`).
`st_atime_`	Time of the last access of the file.
`st_mtime_`	Time of the last modification of the file.
`st_ctime_`	Time of the creation of the file.
`st_size_`	Size of the file in bytes.
`st_blksize_`	Block size for the file.
`st_blocks_`	Number of blocks.
`st_flags_`	File flags.

New HostTimeType

The host control API defines `HostTimeType` as a platform-independent version of the standard C library `time_t` type.

```
typedef long HostTimeType;
```

New HostTmType

The host control API defines `HostTmType` for use in time functions.

```
struct HostTmType
{
  long    tm_sec_;
  long    tm_min_;
  long    tm_hour_;
  long    tm_mday_;
  long    tm_mon_;
  long    tm_year_;
  long    tm_wday_;
```

```
  long    tm_yday_;
  long    tm_isdst_;
};
typedef struct HostTmType HostTmType;
```

HostTmType Fields

`tm_sec_`	Seconds after the minute: range from 0 to 59.
`tm_min_`	Minutes after the hour: range from 0 to 59.
`tm_hour_`	Hours since midnight: range from 0 to 23.
`tm_mday_`	Day of the month: range from 1 to 31.
`tm_mon_`	Months since January: range from 0 to 11.
`tm_year_`	Years since 1900.
`tm_wday_`	Days since Sunday: range from 0 to 6.
`tm_yday_`	Days since January 1: range from 0 to 365.
`tm_isdst_`	Daylight savings time flag.

New HostUTimeType

The host control API defines `HostUTimeType`for use in time functions.

```
struct HostUTimeType
{
  HostTimeType crtime_;
  HostTimeType actime_;
  HostTimeType modtime_;
};
typedef struct HostUTimeType HostUTimeType;
```

HostUTimeType Fields

`crtime_`	Creation time.
`actime_`	Access time.
`modtime_`	Modification time.

Functions

This section describes the host control API functions.

NOTE: For host control API functions that return pointers to character strings (that is, functions that return type `char *`), the returned value is valid only until the next call to a function that returns a pointer to a character string. If you need ongoing access to a character string, you should make a copy of the string before making the subsequent host control function call.

New HostAscTime

Purpose Returns a character string representation of the time encoded in `time`.

Prototype

```
char* HostAscTime(const HostTmType* time);
```

Parameters

`time`	The time structure.

Result The time as a character string.

New HostClock

Purpose Returns an elapsed time.

Prototype

```
HostClockType HostClock(void);
```

Parameters None.

Result The elapsed time in terms of the operating system's clock function (usually the number clock ticks that have elapsed since the start of the process), or -1 if the function call was not successful.

New HostCloseDir

Purpose Closes a directory.

Prototype
```
long HostCloseDir(HostDIRType* directory);
```

Parameters `directory` The directory to be closed.

Result Returns `0` if the operation was successful, and a non-zero value if not.

New HostCTime

Purpose Converts the calendar time in `*timeofday` to a text representation.

Prototype
```
char* HostCTime(const HostTimeType* timeofday)
```

Parameters `timeofday` The calendar time.

Result The calendar time as a time string.

HostErrNo

Purpose Returns 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.

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

`gestSel`

`response`

New HostGetDirectory

Purpose Gets a directory, in support of the operating system file chooser dialog box.

Prototype
```
const char* HostGetDirectory(const char* prompt,
const char* defaultDir);
```

Parameters

`prompt`

`defaultDir` The default directory to get.

Result Returns the directory as a character string.

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.

New HostGetFile

Purpose Gets a file, in support of the operating system file chooser dialog box.

Prototype
```
const char* HostGetFile(const char* prompt,
const char* defaultFile)
```

Parameters

`prompt`

`defaultFile` The default fileto get.

Result Returns the file as a character string.

HostGetFileAttr

Purpose Get the attribute settings of a file or directory. This function can tell you whether the file is read-only, hidden, or a system file.

Prototype

```
long HostGetFileAttr(const char* fileOrPathName,
long* attrFlag)
```

Parameters

`fileOrPathName` The file name or directory path for which you want to get the file attribute setting.

`attrFlag` One of the following attribute flags:

- `hostFileAttrReadOnly`
- `hostFileAttrHidden`
- `hostFileAttrSystem`

The file attribute flags match the EmFileAttr flags:

```
enum
{
   hostFileAttrReadOnly = 1,
   hostFileAttrHidden = 2,
   hostFileAttrSystem = 4
}
```

Result The file attribute.

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 Retrieves 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 2.1.

Table 2.1 Palm OS Emulator preferences file names and locations

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

See Also The HostSetPreference function.

New HostGMTime

Purpose Returns time structure representation of the time, expressed as Universal Time Coordinated, or UTC (UTC was formerly Greenwich Mean Time, or GMT).

Prototype
```
HostTmType* HostGMTime(const HostTimeType* time);
```

Parameters time

Result The time structure.

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

`info`	A `HostGremlinInfo` structure with information about the new horde of gremlins

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.

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

New HostLocalTime

Purpose Returns time structure representation of the time, expressed as local time.

Prototype
```
HostTmType* HostLocalTime(const HostTimeType* time);
```

Parameters

`time`	The time structure.

Result The time structure.

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.

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.

New HostMkDir

Purpose Creates a directory.

Prototype
```
long HostMkDir(const char* directory);
```

Parameters

`directory`	The directory to create.

Result Returns `0` if the operation was successful, and a non-zero value if not.

New HostMkTime

Purpose Alters the parameter values to represent an equivalent encoded local time, but with the values of all members within their normal ranges.

Prototype
```
HostTimeType HostMkTime(HostTmType* time)
```

Parameters

`time`	The time structure.

Result Returns the calendar time equivalent to the encoded time, or returns a value of -1 if the calendar time cannot be represented

New HostOpenDir

Purpose Opens a directory.

Prototype
```
HostDIRType* HostOpenDir(const char* directory);
```

Parameters

`directory`	The directory to open.

Result Returns a directory structure.

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.

Comments This function is available only in the profiling version of the emulator.

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 Boolean 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.

Comments This function is available only in the profiling version of the emulator.

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.

Comments This function is available only in the profiling version of the emulator.

New HostProfileGetCycles

Purpose Returns the current running CPU cycle count.

Prototype
```
long HostProfileGetCycles(void)
```

Parameters None.

Result Returns the current running CPU cycle count.

Comments This function is available only in the profiling version of the emulator.

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. This parameter determines the size of the array used to keep track of function calls. A typical value for `maxCalls` is 65536.
`maxDepth`	The maximum profiling depth. This parameter determines the size of the array used to keep track of function call depth. A typical value for `maxDepth` is 200.

Result Returns `0` if the operation was successful, and a non-zero value if not.

Comments This function is available only in the profiling version of the emulator.

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.

Comments This function is available only in the profiling version of the emulator.

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.

Comments This function is available only in the profiling version of the emulator.

New HostPutFile

Purpose Writes a file, in support of the operating system "Save As" dialog box.

Prototype
```
const char* HostPutFile(const char* prompt, const
char* defaultDir, const char* defaultName);
```

Parameters

`prompt`	
`defaultDir`	The default directory to use.
`defaultName`	The default file name to use.

Result Returns the file name as a character string.

New HostReadDir

Purpose Reads a directory.

Prototype
```
HostDirEntType* HostReadDir(HostDIRType*
directory);
```

Parameters

`directory`	The directory to read.

Result Returns a character array for the directory.

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.

New HostRmDir

Purpose Removes a directory.

Prototype
```
long HostRmDir(const char* directory);
```

Parameters

`directory`	The directory to remove.

Result Returns `0` if the operation was successful, and a non-zero value if not.

New HostSaveScreen

Purpose Saves the LCD frame buffer to the given file name.

Prototype
```
HostErrType HostSaveScreen(const char* fileName)
```

Parameters

`fileName`	The name of the file to which the current LCD frame buffer is to be saved.

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.

IMPORTANT: 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.

IMPORTANT: 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.

New HostSetFileAttr

Purpose Set the attribute settings of a file or directory. This function can set the read-only, hidden, or system-file attribute for the file or directory.

Prototype

```
long HostSetFileAttr(const char* fileOrPathName,
long* attrFlag)
```

Parameters

`fileOrPathName` The file name or directory path for which you want to set the file attribute setting.

`attrFlag` One of the following attribute flags:

- `hostFileAttrReadOnly`
- `hostFileAttrHidden`
- `hostFileAttrSystem`

The file attribute flags match the EmFileAttr flags:

```
enum
{
   hostFileAttrReadOnly = 1,
   hostFileAttrHidden = 2,
   hostFileAttrSystem = 4
}
```

Result The file attribute.

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 2.1.

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 Palm OS Emulator.

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.

IMPORTANT: This function is defined for external RPC clients to call, and returns an error if you call it from within a Palm application.

HostSignalWait

Purpose Waits for a signal from Palm OS Emulator, 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 number of the signal that occurred.

Result Returns `0` if the operation was successful, and a non-zero value if not. Returns the number of the signal that occurred in `signalNumber`.

Comments Palm OS Emulator waits to be restarted after issuing a signal to allow external scripts to perform operations.

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.

New HostSlotHasCard

Purpose Ask whether Emulator is emulating a Virtual File System card for a specific slot number.

Prototype

```
HostBoolType HostSlotHasCard(long slotNo)
```

Parameters

`slotNo`	The slot number. This number can be in the range from 1 up to and including the number returned by function HostSlotMax.

Result A Boolean value that indicates whether Emulator is emulating a Virtual File System card in the slot specified by `slotNo`. This function is provided in support of Expansion Manager emulation.

Comments This function may return `FALSE` if the user has not selected to emulate a Virtual File System card in the given slot, or if Emulator is emulating a different kind of card in that slot.

New HostSlotMax

Purpose Returns the number of Virtual File System cards that Emulator is emulating.

Prototype

```
long HostSlotMax(void)
```

Parameters None.

Result A `long` value indicating the number of Virtual File System cards Emulator is emulating. This function is provided in support of Expansion Manager emulation.

Comments The functions that accept card numbers, HostSlotHasCard and HostSlotRoot, accept numbers from 1 up to and including the number returned by HostSlotMax.

New HostSlotRoot

Purpose Returns a string representing the root directory of the emulated slot.

Prototype

```
const char* HostSlotRoot(long slotNo)
```

Parameters

`slotNo`	The slot number. This number can be in the range from 1 up to and including the number returned by function HostSlotMax.

Result The character string representing the directory to be used as the root for the given Virtual File System card. This function is provided in support of Expansion Manager emulation.

Comments The string returned is in host path format. This function may return NULL if there is no Virtual File System card mounted in the slot specified by `slotNo` or if the user has not selected a root directory for that slot.

New HostStat

Purpose Returns status information about a file.

Prototype

```
long HostStat(const char* filename, HostStatType*
buffer);
```

Parameters

filename	The name of the file or directory for which you want status information
buffer	The structure that stores the status information

Result Returns `0` if the operation was successful, and a non-zero value if not.

New HostStrFTime

Purpose Generates formatted text, under the control of the format parameter and the values stored in the time structure parameter.

Prototype

```
HostSizeType HostStrFTime(char* string,
HostSizeType size, const char* format, const
HostTmType* time)
```

Parameters

`string`	The formatted text
`size`	The size of an array element in the formatted text
`format`	The format definition
`time`	A time structure

Result Returns the number of characters generated, if the number is less than the `size` parameter; otherwise, returns zero, and the values stored in the array are indeterminate.

New HostTime

Purpose Returns the current calendar time.

Prototype
```
HostTimeType HostTime(HostTimeType* time);
```

Parameters

`time`	The time structure.

Result Returns the current calendar time if the operation is successful, and returns -1 if not.

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.

HostTraceClose

Purpose Closes the connection to the external trace reporting tool.

Prototype

```
void HostTraceClose(void);
```

Parameters None.

Result None.

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.

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.

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 2.2 shows the flag types that you can use in the format specification for the tracing output functions.

Table 2.2 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 2.3 shows the output types that you can use in the format specification for the tracing output functions.

Table 2.3 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.

New HostTruncate

Purpose Extends or truncates the file associated with the file handle to the length specified by the size.

Prototype

```
long HostTruncate(const char* filename, long
filesize);
```

Parameters

`filename`	The name of the file.

	`filesize`	The size of the file.

Result Returns the value 0 if the file is successfully changed, or returns -1 if there was an error.

New HostUTime

Purpose Sets the modification time for a file.

Prototype

```
long HostUTime (const char* filename,
HostUTimeType* buffer);
```

Parameters

	filename	The filename of the file.
	buffer	The stored time values.

Result Returns 0 if the file-modification time was successfully changed, or returns -1 if there was an error.

Reference Summary

The tables in this section summarize the host control API functions.

Host Control Database Functions

Table 2.4 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.
`HostSaveScreen`	Saves the LCD frame buffer to a file.

New Host Control Directory Handler Functions

Table 2.5 Host Control Directory Handler Functions

Function	Description
HostCloseDir	Closes a directory.
HostMkDir	Makes a directory.
HostOpenDir	Opens a directory.
HostReadDir	Reads a directory.
HostRmDir	Removes a directory.

Host Control Environment Functions

Table 2.6 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.

New Host Control File Chooser Support Functions

Table 2.7 Host Control File Choose Support Functions

Function	Description
HostGetDirectory	Gets a directory, in support of the operating system file chooser dialog box.
HostGetFile	Gets a file, in support of the operating system file chooser dialog box.
HostPutFile	Writes a file, in support of the operating system file chooser dialog box.

Host Control Gremlin Functions

Table 2.8 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 2.9 Host Control Logging 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 2.10 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 2.11 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.
HostProfileGetCycles	Returns the current running CPU cycle count.
HostProfileInit	Initializes and enables profiling in the debugging host.
HostProfileStart	Turns profiling on.
HostProfileStop	Turns profiling off.

Host Control RPC Functions

Table 2.12 Host Control RPC Functions

Function	Description
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 2.13 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.
HostFGetC	Returns the character at the current position in the specified file. Returns `EOF` to indicate an error.

Table 2.13 Host Control Standard C Library Functions

Function	Description
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.
HostFTell	Retrieves the current position of the specified file. Returns -1 to indicate an error.

Table 2.13 Host Control Standard C Library Functions

Function	Description
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.

New Host Control Time Functions

Table 2.14 Host Control Time Functions

Function	Description
HostAscTime	Returns a character string representation of the time.
HostClock	Returns an elapsed time.
HostCTime	Converts calendar time to a text representation.
HostGMTime	Returns time structure representation of the time expressed as Universal Time Coordinated (UTC). UTC was formerly Greenwich Mean Time (GMT).

Table 2.14 Host Control Time Functions

Function	Description
HostLocalTime	Returns time structure representation of the time expressed as local time.
HostMkTime	Alters the parameter values to represent an equivalent encoded local time, but with the values of all members within their normal ranges.
HostStrFTime	Generates formatted text, under the control of the format parameter and the values stored in the time structure parameter.
HostTime	Returns the current calendar time.
HostUTime	

Host Control Tracing Functions

Table 2.15 Host Control Tracing Functions

Function	Description
HostTraceClose	Must be called when done logging trace information.
HostTraceInit	Must be called before logging any trace information.
HostTraceOutputT	Outputs text to the trace log using printf-style formatting.
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.

Table 2.15 Host Control Tracing Functions *(continued)*

Function	Description
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.

3

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.

This chapter covers the following topics:

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.

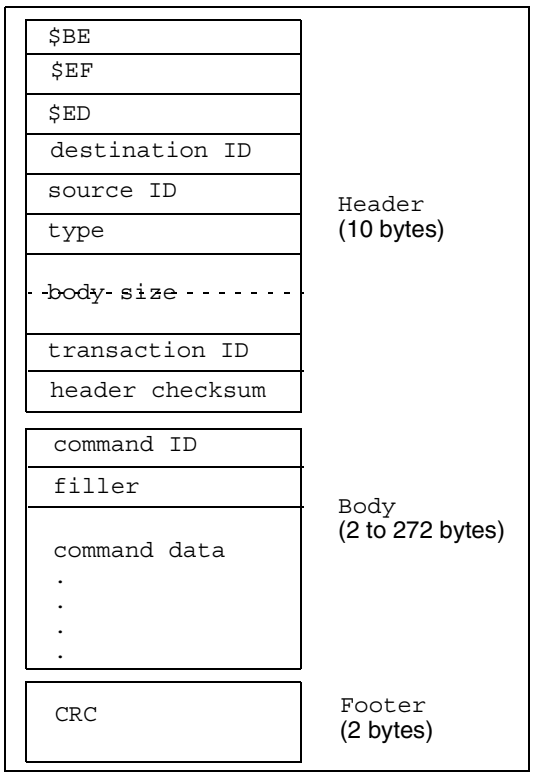
Although the typical flow of packets involves the host sending a request and the target sending back a response, although there are a some exceptions, as follows:

- 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 3.1.

Figure 3.1 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" on page 150 for a description of the structure used to represent the two byte body header (the command and filler bytes), and see Table 3.1 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 3.1 shows the command constants.

Table 3.1 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 3.1 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.

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" on page 149.

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 packet 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 3.2 summarizes the command packets that you can use with the debugger protocol.

Table 3.2 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 3.2 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.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W