Palm OS® Programming Development Tools Guide

Palm OS® 5 SDK (68K) R3

CONTRIBUTORS

Written by Brian Maas
Engineering contributions by Keith Rollin, Ken Krugler, Jesse Donaldson, Andy Stewart, Kenneth Albanowski, and Derek Johnson

Palm OS Programming Development Tools Guide
Document Number 3011-006
November 6, 2003
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

About This Document

Palm OS® Programming Development Tools Guide describes various tools you can use to develop software for Palm Powered™ handhelds.

Palm OS Documentation

In addition to this book, you may be interested in the following Palm OS documentation:

Document	Description
Palm OS Programmer's API Reference	An API reference document that contains descriptions of all Palm OS function calls and important data structures.
Palm OS Programmer's Companion, vol. I and *Palm OS Programmer's Companion*, vol. II, *Communications*	A guide to application programming for the Palm OS. These volumes contain conceptual and "how-to" information that complements *Palm OS Programmer's API Reference*.
Using Palm OS Emulator	A guide to testing applications with Palm OS Emulator, including a reference of the Host Control API functions. The information in this book was previously part of *Palm OS Programming Development Tools Guide*.
Testing with Palm OS Simulator	A guide to testing application with Palm OS Simulator.
Constructor for Palm OS	A guide to creating application interfaces using Constructor for Palm OS.
Palm OS User Interface Guidelines	A guide describing how to design applications for Palm Powered handhelds.

What This Volume Contains

This volume is designed for random access. That is, you can read any chapter in any order. In general, each chapter covers a different

Palm OS development tool, though chapters 2 through 4 discuss topics relating to Palm Debugger.

Here is an overview of this volume:

- Chapter 1, "Using Palm Debugger," on page 1. Provides an introduction to Palm Debugger, which is an assembly language and limited source code level debugger for Palm OS programs. This chapter describes how to use Palm Debugger, including a description of its expression language and a variety of debugging strategies and tips.
- Chapter 2, "Palm Debugger Command Reference," on page 51. Provides a complete reference description for each command available in Palm Debugger.
- Chapter 3, "Debugger Protocol Reference," on page 85. Describes the API for sending commands and responses between a debugging host, such as Palm Debugger, and a debugging target, which can be a Palm Powered handheld ROM or an emulator program such as Palm OS Emulator.
- Chapter 4, "Using the Console Window," on page 113. Describes how the Console Window can be used to perform maintenance and do high-level debugging of a Palm handheld device.
- Chapter 5, "Using Palm Reporter," on page 161. Describes Palm Reporter, which is a trace utility that can be used with Palm OS Emulator.
- Chapter 6, "Using the Overlay Tools," on page 173. Describes how you can create national language versions of your application by creating interface overlays.
- Appendix A, "Resource Tools," on page 183. Provides a short description of resource tools that can be used to develop application resources.
- Appendix B, "Simple Data Types," on page 185. Describes the simple data type name changes made in recent versions of the Palm OS software.

Summary of Changes

- Chapters from the prior edition of this manual ("Using Palm OS Emulator" and "Host Control API") have been moved

from this book into a new manual called *Using Palm OS Emulator*. For more information, see the Palm OS documentation web page.

- Chapter 5, "Using Palm Reporter," on page 161 has been updated to include information on the Macintosh version of Palm Reporter.

Additional Resources

- Documentation

  PalmSource publishes its latest versions of this and other documents for Palm OS developers at

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

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

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

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

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

1

Using Palm Debugger

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

This chapter provides an introduction to and overview of using Palm Debugger. The commands that you can use are described in Chapter 2, "Palm Debugger Command Reference."

This chapter contains the following sections:

- "About Palm Debugger" on page 2 provides a broad overview of the program and a description of its windows.
- "Connecting to The Handheld Device" on page 4 describes how to connect Palm Debugger with the Palm OS Emulator or with a Palm Powered™ handheld device.
- "Using the Console and Debugging Windows Together" on page 8 describes how to use the menus and keyboard to send commands to the handheld device from the debugging and console windows.
- "Using the Debugging Window" on page 15 and "Using the Source Window" on page 32 describe the command and display capabilities available in each of these windows. The debugging window section also includes a full description of "Using Debugger Expressions" on page 17.
- "Palm Debugger Error Messages" on page 36 describes how to decode the error messages you receive from Palm Debugger.
- "Palm Debugger Tips and Examples" on page 37 provides a collection of tips to make your debugging efforts easier and examples of performing common debugging tasks.

About Palm Debugger

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

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

NOTE: You can use Palm Debugger with a Palm Powered handheld device, or with the Palm OS Emulator program. Debugging is the same whether you are sending commands to the emulator or to actual hardware. Connecting with either a handheld device or the Emulator is described in "Connecting Palm Debugger With a Target" on page 3.

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

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

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

- The debugging interface is provided by the *debugger nub* on the handheld device. You can attach to the debugger nub and then send debugging commands to the debugger nub from Palm Debugger's debugging window. For more information about using the debugging window and the debugging commands, see "Using the Debugging Window" on page 15.

The console window and the debugging window each has its own set of commands that you can use to interface with the handheld device. The debugging commands are described in Chapter 2, "Palm Debugger Command Reference," and the console window commands are described in Chapter 4, "Using the Console Window."

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

On certain platforms, Palm Debugger also provides a multi-pane source window for source-level debugging. You can use this window if you have compiled your program with certain compilers that generate an appropriate symbol file. Table 1.1 summarizes the Palm Debugger windows.

Table 1.1 Palm Debugger Windows

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

Connecting Palm Debugger With a Target

You can use Palm Debugger to debug programs running on a Palm Powered handheld device or to debug programs running on a

hardware emulator such as the Palm OS Emulator. This section describes how to connect the debugger to each of these targets.

Connecting to The Palm OS Emulator

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

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

Connecting to The Handheld Device

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

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

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

Activating Debugging Input

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

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

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

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

1. Use the to enter debugger mode on the handheld device, as described in "Using Shortcut Numbers to Activate the Windows" on page 6.
2. If you have already activated the console nub, you can use the **Break** command in the Source menu to activate the debugger nub. The **Break** command sends a key command to the handheld device that is identical to using the sequence.
3. Compile a `DbgBreak()` call into your application, and run the application until you encounter that call.

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

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

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

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

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

Verifying Your Connection

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

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

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

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

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

Using Shortcut Numbers to Activate the Windows

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

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

To enter a shortcut number, follow these steps:

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

2. Next, tap the stylus twice, to generate a dot (a period).
3. Next, draw a number character in the number entry portion of the device's text entry area. Table 1.2 shows the different shortcut numbers that you can use.

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

 .2

Table 1.2 Shortcut Numbers for Debugging

Shortcut	Description	Notes
	The device enters debugger mode, and waits for a low-level debugger to connect. A flashing square appears in the top left corner of the device.	This mode opens a serial port, which drains power over time. You must perform a soft reset or use the debugger's `reset` command to exit this mode.

Table 1.2 Shortcut Numbers for Debugging *(continued)*

Shortcut	Description	Notes
.2	The device enters console mode, and waits for communication, typically from a high-level debugger.	This mode opens a serial port, which drains power over time. You must perform a soft reset to exit this mode.
.3	The device's automatic power-off feature is disabled.	You can still use the device's power button to power it on and off. Note that your batteries can drain quickly with automatic power-off disabled. You must perform a soft reset to exit this mode.

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

Using the Console and Debugging Windows Together

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

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

- `reset`

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

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

Entering Palm Debugger Commands

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

Table 1.3 summarizes the keyboard commands that you can use to enter commands in Palm Debugger's console or debugging windows.

Table 1.3 Entering Palm Debugger Commands From the Keyboard

Command description	Windows key(s)	Macintosh key(s)
Execute selected text as command(s). You can select multiple lines to sequentially execute multiple commands.	Enter	Enter on numeric keypad, or CMD+RETURN
Execute the current line (no text selected).		
Display help for a command	`Help <cmdName>`	`Help <cmdName>`
Enter a new line without executing the text	SHIFT+ENTER	Return
Copy selected text from window to clipboard	CTRL+C	CMD+C

Table 1.3 Entering Palm Debugger Commands From the Keyboard *(continued)*

Command description	Windows key(s)	Macintosh key(s)
Paste clipboard contents to window	CTRL+V	CMD+V
Cut selected text from window to clipboard	CTRL+X	CMD+X
Delete previous command's output from the window	CTRL+Z	Not available
Delete all text to the end	SHIFT+Backspace	CMD+DELETE

Palm Debugger Menus

Palm Debugger includes five menus, as summarized in Table 1.4. The most commonly used menu commands are on the Connection and Source menus; these commands are described in other sections in this chapter.

Table 1.4 Palm Debugger Menu Commands

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

Table 1.4 Palm Debugger Menu Commands *(continued)*

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

Palm Debugger Command Syntax

Palm Debugger's help facility uses simple syntax to specify the format of the commands that you can type in the console and debugging windows. This same syntax is used in Chapter 2, "Palm

Debugger Command Reference." This section summarizes that syntax.

The basic format of a command is specified as follows:

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

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

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

The following is an example of a command definition

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

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

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

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

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

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

Specifying Command Options

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

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

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

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

Specifying Numeric and Address Values

Many of the debugging commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (#) or binary (%). Table 1.5 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 1.5 Specifying Numeric Values in Palm Debugger

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

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

For more information, see "Specifying Constants" on page 17.

Using the Debugging Window

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

The debugging window provides numerous capabilities, including the following:

- A rich expression language for specifying command arguments, as described in "Using Debugger Expressions" on page 17.
- Ability to debug applications, system code, extensions, shared libraries, background threads, and interrupt handlers.
- Custom aliases for commands or groups of commands, as described in "Defining Aliases" on page 31.
- Script files for saving and reusing complex sequences of commands, as described in "Using Script Files" on page 31.
- Templates for defining data structure layouts in memory, which allow you to view a structure with the memory display commands. Templates are described in "Defining Structure Templates" on page 29.
- Your aliases and templates can be saved in files that are automatically loaded for you when Palm Debugger starts execution, as described in "Automatic Loading of Definitions" on page 31.

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

- See "Displaying Registers and Memory" on page 23 for examples of using the debugging commands to display the current register values.
- See "Using the Flow Control Commands" on page 25 for examples of using commands to set breakpoints.
- See "Using the Heap and Database Commands" on page 28 for examples of using commands to examine the heap and databases.

The remainder of this section describes how to use these capabilities. Table 1.6 shows the most debugging window command categories.

Table 1.6 Debugging Window Command Categories

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

All of the debugging commands are described in detail in Chapter 2, "Palm Debugger Command Reference."

Before you can use the debugging commands, you must attach Palm Debugger to the debugger nub on the handheld device, as described in "Activating Debugging Input" on page 4.

Using Debugger Expressions

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

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

Specifying Constants

The expression language lets you specify numbers as character constants.

Character Constants

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

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

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

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

Binary Numbers

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

```
%00111000
%1010
```

The size of the resulting value is determined as follows:

Number of Digits	Result Type
1 to 8	`UInt8`
8 to 16	`UInt16`
more than 16	`UInt32`

Decimal Numbers

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

```
#256
#32756
```

Hexadecimal Numbers

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

```
c123
C123
F0
$A0
```

The size of the resulting value is determined as follows:

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

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

Using Operators

Palm Debugger expression language includes the typical set of binary and unary operators, as summarized in Table 1.7.

Table 1.7 Palm Debugger Expression Language Operators

Type	Operator	Description	Example
Cast	`.a`	Casts the value to an address.	`0ff0.a`
	.b	Casts the value to a byte.	45.b
	`.l`	Casts the value to a double word.	`45.l`
	`.w`	Casts the value to a word.	`45.w`
	`.s`	Extends the sign of its operand without changing the operand's size.	`45.s`
Unary	`~`	Performs a bitwise NOT of the operand.	`~1`
	`-`	Changes the sign of the operand.	`2*-1`
Dereference	`@`	Dereferences an address or integer value. See Table 1.8 for more examples.	@A7
Arithmetic	`*`	Multiplies the two operands together.	`A1*2`
	`/`	Divides the first operand by the second operand.	`21/3`
	`+`	Adds the two operands together.	`A2+2`
	`-`	Subtracts the second operand from the first operand.	A2-2

Table 1.7 Palm Debugger Expression Language Operators *(continued)*

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

The Dereference Operator

The @ dereference operator is similar to the * dereference operator used in the C programming language. This operators dereferences an address value, as shown in Table 1.8.

Table 1.8 Dereference Operator Examples

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

Register Variables

The expression language provides named variables for each register. The names of these variables are replaced by their respective register values in any expression. Table 1.9 shows the register name variables.

Table 1.9 The Built-in Register Variables

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

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

For example, the following expression:

```
a0+d0
```

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

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

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

Special Shortcut Characters

Palm Debugger's expression language includes the two special value characters show in Table 1.10. These characters are converted into values in any expression.

Table 1.10 Special Value Expression Characters

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

Performing Basic Debugging Tasks

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

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

The final section of this chapter, "Palm Debugger Tips and Examples" on page 37, provides examples of how to perform other debugging tasks.

Assigning Values to Registers

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

```
pc=010c8954
```

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

```
pc = 010c8954c
```

Displaying Registers and Memory

One of the primary operations you perform with a debugger is to examine and change values in memory. Palm Debugger provides a number of commands for displaying registers, memory locations, the program counter, and the stack. Table 1.11 summarizes the commands you commonly use to examine memory and related values.

Table 1.11 Frequently Used Memory Commands

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

Table 1.11 Frequently Used Memory Commands *(continued)*

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

Palm Debugger also lets you define structure templates and use those for displaying memory values. For example, you can define a structure that matches the layout of a complex data structure, and then display that structure with a single `dm` command. For more information about structure templates, see "Defining Structure Templates" on page 29.

Listing 1.1 shows an example of displaying memory with the `dm` command and disassembling memory with the `il` command. It also provides several examples of using expressions with these commands. In this example, **`boldface`** is used to denote commands that you type, and <= starts a comment.

Listing 1.1 Displaying and Disassembling Memory

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

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

dm #100                 <=Display memory at address 100 decimal
00000064: 10 C6 BE 32 10 C6 BE 60 10 C6 BE 8E 10 C6 BE BC "...2...`........"

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

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

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

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

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

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

All of the commands mentioned in this section are described in detail in Chapter 2, "Palm Debugger Command Reference."

Using the Flow Control Commands

Palm Debugger provides a number of commands for setting breakpoints and continuing the flow of execution. Table 1.12 summarizes the commands you commonly use for these purposes.

Table 1.12 Commonly Used Flow Control Commands

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

Listing 1.2 shows an example of setting breakpoints, disassembling, and using other flow control commands to debug an application. In this example, **`boldface`** is used to denote commands that you type, and <= starts a comment.

Listing 1.2 Using the Debugging Flow Control Commands

```
sc                <= Display stack crawl, listed from oldest to newest. In this
                  <= example, the current fcn was called from EventLoop+0016
Calling chain using A6 Links:
 A6 Frame   Caller
 00000000  10C68982  cjtkend+0000
 00015086  10C6CA26  __Startup__+0060
 00015066  10C6CCCE  PilotMain+0250
 00014FC2  10C0F808  SysAppLaunch+0458
 00014F6E  10C10258  PrvCallWithNewStack+0016
 00013418  10CD88B2  __Startup__+0060
 000133F8  10CDB504  PilotMain+0036
 000133DE  10CDB47C  EventLoop+0016

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

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

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

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

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

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

cl                  <= Clear all breakpoints
```

```
All breakpoints cleared

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

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

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

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

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

All of the commands mentioned in this section are described in detail in Chapter 2, "Palm Debugger Command Reference."

Using the Heap and Database Commands

You can use the heap and database commands to display information about the databases and heaps on the handheld device. These commands, which are summarized in Table 1.13, mirror commands available from the console window.

Table 1.13 Commonly Used Heap and Database Commands

Command	Description
`dir`	Lists the databases.
`hchk`	Checks a heap.
`hd`	Displays a dump of a memory heap.

Table 1.13 Commonly Used Heap and Database Commands *(continued)*

Command	Description
`hl`	Lists all of the memory heaps on the specified memory card.
`ht`	Performs a heap summary.

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

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

All of the commands mentioned in this section are described in detail in Chapter 2, "Palm Debugger Command Reference."

To learn more about the console window and all of the console commands, see Chapter 4, "Using the Console Window."

Advanced Debugging Features

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

- defining structure template for displaying memory
- defining aliases for commands
- using script files to run sequences of commands
- automated loading of structure and alias definitions at program start-up time

Defining Structure Templates

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

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

Table 1.14 summarizes the commands you use to define and display templates. For more information about these commands, see Chapter 2, "Palm Debugger Command Reference."

Table 1.14 Structure Template Commands

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

Note that the structure and field names must be quoted in your structure template definition commands. Listing 1.3 shows the debugging commands used to define a template named `PointType`, and then defines a second template named `RectangleType` that uses two `PointType` fields.

Listing 1.3 Defining and using two structure templates

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


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

sizeof PointType
```

```
Size = 4 byte(s)

sizeof RectangleType
Size = 8 byte(s)

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

Defining Aliases

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

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

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

Using Script Files

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

```
run "MyCommands"
```

Automatic Loading of Definitions

When Palm Debugger is launched, it automatically runs the script file named `UserStartupPalmDebugger`. You can store your

aliases, script files, and data structure templates in this file to have them available whenever you use Palm Debugger.

Using the Source Window

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

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

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

The source window is split into two panes:

- The upper pane displays the values of local variables for the current function.
- The lower pane displays the source code. This pane is automatically updated whenever you move through your code with flow control commands. You can also scroll this pane to view the code or to set a breakpoint.

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

  – a solid red circle is displayed next to a line that contains a breakpoint

  – a green arrow is displayed next to the line containing the current program location

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

Debugging With the Source Window

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

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

1. Activate the console nub, as described in "Activating Console Input" on page 114.
2. Select **Install Database and Load Symbols** from the Source menu.
3. Select the `PRC` file to load onto the device.
4. Palm Debugger imports the `PRC` file into the handheld device and looks in the same directory for the associated symbol file.

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

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

Using Symbol Files

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

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

A symbol file contains the following items:

- the names of each of the source files that were linked together to create the code resource

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

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

Using the Source Menu

Palm Debugger's Source menu contains commands that you can use for source level debugging. Table 1.15 summarizes these commands. Note that several of these commands are also available from the Source context menu, as described in the next section.

Table 1.15 Source Menu Commands

Command	Description
Break	Halts the handheld device in the debugger nub by sending the same key event as does the . The device must be running the console nub to activate this command.
Step Into	Single steps one source line, and stops if it steps into a subroutine.
Step Over	Single steps one source line. If it steps into a subroutine, doesn't stop until the subroutine returns.

Table 1.15 Source Menu Commands *(continued)*

Command	Description
Go	Continues execution until a breakpoint is encountered.
Go Till	Sets a temporary breakpoint at the currently selected line in the source window and then continues execution.
Toggle Breakpoint	Toggles a breakpoint on or off at the currently selected line in the source window.
Disassemble at Cursor	Disassembles code at the currently selected line in the source window. The disassembled output is displayed in the debugging window.
Show Current Location	Scrolls the source window to show the current line in the source file.
Install Database and Load Symbols	Imports a `PRC` file into the handheld device and looks in the same directory for the associated symbol file.
Load Symbols	Opens a symbol file for use by Palm Debugger.
Remove All Symbols	Unloads any loaded symbols.

Using the Source Window Context Menu

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

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

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

Source Window Debugging Limitations

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

- You cannot display a stack crawl in the source window. You need to switch to the debugging window and use the `sc` command.
- Local variables that are structures or pointers to structures display as hexadecimal addresses in the local variables pane of the source window. To view the contents of these structures, you need to use the `dm` command in the debugging window.
- You cannot view global variables in the source window.
- Local variables are only displayed in hexadecimal format.
- You cannot change the values of local variables from the source window. To change these values, you must use the `sb`, `sw`, or `sl` commands in the debugging window.

Palm Debugger Error Messages

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

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

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

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

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

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

Palm Debugger Tips and Examples

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

- "Performing Calculations"
- Saving time with "Shortcut Characters" and "Repeating Commands" on page 38
- "Finding a Specific Function" on page 38
- "Finding Memory Corruption Problems" on page 42
- "Displaying Local Variables and Function Parameters" on page 44
- "Changing the Baud Rate Used by Palm Debugger" on page 47
- "Debugging Applications That Use the Serial Port" on page 48
- "Importing System Extensions and Libraries" on page 48
- "Determining the Current Location Within an Application" on page 49

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

Performing Calculations

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

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

Shortcut Characters

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

Repeating Commands

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

Finding a Specific Function

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

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

- Use the source level debugging support to locate your routine.

Rebuilding the Application

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

Setting an A-trap Break

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

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

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

```
atb "evtgetevent"
A-trap set on 011d (evtgetevent)
g
Remote stopped due to: A-TRAP BREAK EXCEPTION
'EvtGetEvent'
+$0000 10C3B1E2 *LINK A6,$0000 | 4E56 0000
```

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

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

```
gt @sp
EXCEPTION ID = $80
```

```
'EventLoop'
+$0016 1001B2E6 *MOVE.L A2,-(A7) | 2F0A
```

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

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

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

The `atb`, `g`, `gt`, and `il` commands are described in detail in Chapter 2, "Palm Debugger Command Reference".

Using the Find Text Command

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

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

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

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

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

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

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

You can ensure that the routine you've found is the one you want by disassembling the current routine with the `il` command and searching through the routine with the `ft` command.

NOTE: When you use the `ft` command, the first instance of the search string is actually a copy of the search string the debugger nub is using. You must search a second time to find the first "actual" instance of the text string.

Using the Source Level Debugging Support

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

1. Launch the console nub on the handheld device, as described in "Activating Console Input" on page 114.
2. Open your symbols file. You can use the **Open Symbol File** command from Palm Debugger's Source menu.
3. After the symbol file has loaded, choose the **Break** command from the Source menu to break into the debugger nub on the device.
4. In the source window, select the source line of the routine you want to debug.
5. Select **Toggle Breakpoint** from the Source menu to set the breakpoint.

Finding Memory Corruption Problems

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

- heap corruptions
- application variable corruption

Tracking Down Heap Corruption

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

```
hchk 0
Heap OK
```

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

```
hd 0

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

```
...

000114E8 -------- F00114E8 000FF8 001008  #0  #0    FM
-000124F0 -------- F00124F0 001000 001008  #0 #15    fM
-00017D30 -------- F0017D30 00003C 000044  #0 #15    fM SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15    fM Feature Manager Globals
(FtrGlobalsType)
-00017D84 -------- F0017D84 000024 00002C  #0 #15    fM DmOpenInfoPtr: 'Update
3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15    fM DmOpenRef: 'Update
3.0.2'
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15    fM Handle Table: 'Ô©Update
3.0.2'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15    fM DmOpenInfoPtr:
'Ô©Update 3.0.2'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15    fM DmOpenRef: 'Ô©Update
3.0.2'
--------------------------------------------------------------------------------
-------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010C50 bytes)
  Movable Chunks:     #51     (005E80 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

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

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

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

```
Minimum dynamic heap free space recording OFF
```

Note that the memory checking modes can seriously degenerate the performance of an application. You can enable or disable various `mdebug` options to strike a balance between performance and debugging information. For more information, see "mdebug" on page 145.

The `hd`, `hchk`, and `mdebug` commands are described in detail in Chapter 2, "Palm Debugger Command Reference."

Tracking Down Global Variable Corruption

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

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

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

Displaying Local Variables and Function Parameters

If you are debugging with the source window, the current function's local variables and parameters are displayed in the upper pane of the window. However, if you do not have access to symbol information, you need to use debugging commands to manually look up the variable values. This section describes the steps you need to take to look up values for a typical function, which is shown in Listing 1.4

Listing 1.4 An Example Function for Viewing Local Variables and Parameters

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

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

  // body of function omitted for clarity
  ...

  return false;
}
```

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

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

The first `UInt32` on the stack upon function entry is the return address for the function. Immediately following that are the parameter values, from left to right. In the listing above, if you

display the memory pointed to by the stack pointer at the beginning of the function, you see the following:

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

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

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

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

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

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

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

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

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

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

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

Changing the Baud Rate Used by Palm Debugger

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

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

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

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

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

Debugging Applications That Use the Serial Port

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

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

Making Sure the Baud Rates Match

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

Importing System Extensions and Libraries

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

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

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

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

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

Determining the Current Location Within an Application

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

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

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

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

```
sc
Calling chain using A6 Links:
A6 Frame Caller
00000000 10C68982 cjtkend+0000
```

```
00015086 10C6CA26 __Startup__+0060
00015066 10C6CCCE PilotMain+0250
00014FC2 10C0F808 SysAppLaunch+0458
00014F6E 10C10258 PrvCallWithNewStack+0016
0001491E 1001CC7E start+006E
000148E6 1001CF44 PilotMain+001C
```

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

```
opened

name                 resDB     cardNum    accessP         ID    openCnt mode
-------------------------------------------------------------------------------
LauncherDB           no          0        00015146  0001814F    1    0003
*Launcher            yes         0        00016DD2  00D1FA98    1    0001
*Graffiti ShortCuts  yes         0        00017D5C  001FFE7F    1    0007
*System              yes         0        00017FEE  00D20A44    1    0005
-------------------------------------------------------------------------------
Total: 4 databases opened
```

2

Palm Debugger Command Reference

This chapter describes Palm Debugger commands. For an introduction to using Palm Debugger, see Chapter 1, "Using Palm Debugger."

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

- "Debugging Window Commands" on page 53 provides a reference description for each command that you can use in the debugging window to communicate with the debugger nub running on the handheld device. The command reference listings are ordered alphabetically.
- "Debugging Command Summary" on page 80 provides tables that summarize the debugging commands by category.

Command Syntax

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

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

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

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

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

The following is an example of a command definition

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

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

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

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

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

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

Specifying Command Options

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

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

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

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

Specifying Numeric and Address Values

Many of the debugging commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (#) or binary (%). Table 2.1 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 2.1 Specifying numeric values in Palm Debugger

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

For more information, see "Specifying Constants" on page 17.

Using the Expression Language

When you send commands from the debugging window to the debugger nub on the handheld device, you can use Palm Debugger's expression language to specify the command arguments. This language is described in "Using Debugger Expressions" on page 17.

Debugging Window Commands

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

NOTE: You can use Palm Debugger's expression language to specify arguments to debugging window commands. The expression language is described in "Using Debugger Expressions" on page 17.

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

Table 2.2 Debugging window command categories

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

>

Purpose Defines a structure field.

Usage

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

Parameters

`typeName` The type of the field.

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

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

Example

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

alias

Purpose Defines or displays an alias.

Usage

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

Parameters

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

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

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

Example

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

aliases

Purpose Displays the names of all defined aliases.

Usage

```
aliases
```

Parameters None.

Example

```
aliases
ls
```

atb

Purpose Adds an A-Trap break.

Usage

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

Parameters

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

atc

Purpose Clears an A-Trap break.

Usage

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

Parameters

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

atd

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

Usage

```
atd
```

Parameters None.

atr

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

Usage

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

Parameters

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

att

Purpose Attach to the handheld device.

Usage

```
att [options]
```

Parameters

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

Example

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

NOTE: The `att` command will not connect Palm Debugger to Palm OS Simulator. Instead, you should connect from Palm OS Simulator to Palm Debugger by either:

- Entering "shortcut . 1" as described in "Using Shortcut Numbers to Activate the Windows" on page 6 from Palm OS Simulator.

- Entering CTRL+PAUSE (or CTRL+ATTN) from Palm OS Simulator.

Either of these methods will cause Palm OS Simulator to enter debug mode. Next, use the PalmDebugger command `g` to resume debugging.

bootstrap

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

Usage

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

Parameters

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

br

Purpose Sets a breakpoint at the specified address.

Usage

```
br [options] <addr>
```

Parameters

`options`	Optional. You can specify the following option: `\toggle` Toggles the breakpoint on or off.

`addr`	The memory address at which to set the breakpoint.

brc

Purpose Clears a breakpoint or all breakpoints.

Usage

```
brc
    brc <addr>
```

Parameters

`addr`	A memory address.

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

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

brd

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

Usage

```
brd
```

Parameters None.

cardinfo

Purpose Retrieves information about a memory card.

Usage

```
cardinfo <cardNum>
```

Parameters

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

Comments You can use the `cardinfo` command in either the Console window or the debugging window.

Example

```
cardinfo 0

Name: PalmCard
Manuf: Palm, Inc
Version: 0001
CreationDate: B1243780
```

```
ROM Size: 00118FFC
RAM Size: 00200000
Free Bytes : 0015ACB2
Number of heaps: #3
```

cl

Purpose Clears a breakpoint or all breakpoints.

Usage

```
cl
    cl <addr>
```

Parameters

`addr`	A memory address.

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

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

db

Purpose Displays the byte value at a specified address.

Usage

```
db <addr>
```

Parameters

`addr`	A memory address.

Example

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

dir

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

Usage

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

Parameters

`cardNum`	The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.
`searchOptions`	Optional. Options for listing a specific database. Specify any combination of the following flags.

`\c <creatorID>`
Search for a database by creator ID.

`\latest`
List only the latest version of each database.

`\t <typeID>`
Search for a database by its type.

`displayOptions` Optional. Options for which information is displayed in the listing. Specify any combination of the following flags.

`\a` Show all information.

`\at` Show the database attributes.

`\d` Show the database creation, modification, and backup dates.

`\i` Show the database appInfo and sortInfo field values.

`\id` Show the database chunk ID

`\s` Show the database size

`\m` Show the database modification number.

`\n` Show the database name.

`\r` Show the number of records in the database.

`\tc` Show the database type ID and creator ID.

`\v` Show the database version number.

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

```
dir 0
```

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

```
dir 0 -a
```

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

Example

```
dir 0

name                     ID      total       data
-------------------------------------------------
*System                 00D20A44  392.691 Kb  390.361 Kb
*AMX                    00D209C4   20.275 Kb   20.123 Kb
*UIAppShell             00D20944    1.327 Kb    1.175 Kb
*PADHTAL Library        00D208E2    7.772 Kb    7.674 Kb
*IrDA Library           00D20876   39.518 Kb   39.402 Kb
 ...
 MailDB                 0001817F    1.033 Kb    0.929 Kb
 NetworkDB              0001818B    0.986 Kb    0.722 Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842 Kb
 DatebookDB             000181FB    0.084 Kb    0.000 Kb
-------------------------------------------------
Total: 41
```

dl

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

Usage

```
dl <addr>
```

Parameters

`addr`	A memory address.

Example

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

dm

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

Usage

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

Parameters

`addr`	A memory address.
`count`	Optional. The number of bytes to display.

`template`.	The name of the structure template to use. This defines how much memory to display and how to display it.

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

Example

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

dump

Purpose Dumps memory to a file.

Usage

```
dump <"filename"> <addr> <numBytes>
```

Parameters

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

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

dw

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

Usage

```
dw <addr>
```

Parameters

`addr`	A memory address.

Example

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

dx

Purpose Enables or disables `DbgBreak()` breaks.

```
dx
```

Parameters None.

fb

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

Usage

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

Parameters

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

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

Example

```
fb ff 0100 200
dm 00000110   ;00000110: FF 00 00 00 03 18 00 00   03 BC 00
01 7D 72 00 01 "............}r.."
```

fill

Purpose Fills memory with a specified byte value.

Usage

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

Parameters

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

Example

```
fill 0100 8 FF
```

fl

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

Usage

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

Parameters

`value`	The byte value to find.
`addr`	The address at which to start the search.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags:

`\a`	Find all occurrences within the specified range.
`\i`	Use caseless comparison.

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

Example

```
fl ffff 0 1000

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

ft

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

Usage

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

Parameters

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

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

Example

```
ft "abc" 0 1000

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

fw

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

Usage

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

Parameters

`value`	The value to find.
`addr`	The address at which to start the search.
`numBytes`	The number of bytes to search.
`flags`	Optional. You can specify the following flags:

`\a`	Find all occurrences within the specified range.
`\i`	Use caseless comparison.

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

Example

```
fw 32000 0 1000

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

g

Purpose Continues execution.

Usage

```
g
g <addr>
```

Parameters

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

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

Example

```
g
```

gt

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

Usage

```
gt <addr>
```

Parameters

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

hchk

Purpose Checks the integrity of a heap.

Usage

```
hchk <heapId> [options]
```

Parameters

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

	`options`	Optional. You can specify the following option: `\c` Check the contents of each chunk.

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

Example

```
hchk 0000
Heap OK
```

hd

Purpose Displays a hexadecimal dump of the specified heap.

Usage

```
hd <heapId>
```

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

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

Example

```
hd 0
```

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

...
```

```
000114E8 -------- F00114E8 000FF8 001008  #0  #0    FM
-000124F0 -------- F00124F0 001000 001008  #0 #15    fM
-00017D30 -------- F0017D30 00003C 000044  #0 #15    fM SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15    fM Feature Manager Globals
(FtrGlobalsType)
-00017D84 -------- F0017D84 000024 00002C  #0 #15    fM DmOpenInfoPtr: 'Update
3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15    fM DmOpenRef: 'Update
3.0.2'
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15    fM Handle Table: 'Ô©Update
3.0.2'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15    fM DmOpenInfoPtr:
'Ô©Update 3.0.2'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15    fM DmOpenRef: 'Ô©Update
3.0.2'
--------------------------------------------------------------------------------
-------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010C50 bytes)
  Movable Chunks:     #51     (005E80 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

help

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

Usage

```
help
    help <command>
    ?
    ? <command>
```

Parameters

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

Comments You can use the `help` command in either the Console window or the debugging window.

Example

```
help hchk

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

hl

Purpose Displays a list of memory heaps.

Usage
```
hl <cardNum>
```

Parameters

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

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

You can use the `hl` command in either the Console window or the debugging window.

Example
```
hl 0

   index   heapID  heapPtr     size     free  maxFree    flags
-------------------------------------------------------------
       0     0000 00001480 00016B80 00010C50 0000EC48     8000
       1     0001 1001810E 001E7EF2 0014AD6A 00147D3A     8000
       2     0002 10C08212 00118DEE 0000A01C 0000A014     8001
```

ht

Purpose Displays summary information for the specified heap.

Usage
```
ht 0
```

Parameters None.

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

You can use the `ht` command in either the Console window or the debugging window.

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

il

Purpose Disassembles code in a specified line range.

Usage

```
il [<addr> | <"funcName"> [lineCount]]
```

Parameters

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

`funcName` Optional. The name of the function whose code you want disassembled.

`lineCount` Optional. If you provide a value for `addr`, you can also specify the number of lines of code to disassemble starting at `addr`.

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

Example

```
il 0100

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

info

Purpose Displays information about a memory chunk.

Usage

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

Parameters

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

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

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

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

keywords

Purpose Lists all debugger keywords.

Usage
```
keywords
```

Parameters None.

Example
```
keywords

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

load

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

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

Parameters

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

opened

Purpose Lists all of the currently opened databases.

Usage `opened`

Parameters None.

Comments You can use the `opened` command in either the Console window or the debugging window.

Example `opened`

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

penv

Purpose Displays current environment information for the debugger.

Usage `penv`

Parameters None.

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

Example

```
penv
============================
DebOut = false
SymbolsOn = true
StepRegs = false
ReadMemHack = false
Attached = true
.........................
dot address = 00000000
last address = 00001022
last count = 0000000a
============================
```

reg

Purpose Displays all registers.

Usage
```
reg
```

Parameters None.

Example
```
reg

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

reset

Purpose Performs a soft reset on the handheld device.

Usage
```
reset
```

Parameters None.

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

You can use the `reset` command in either the Console window or the debugging window.

Example
```
reset
Resetting system
```

run

Purpose Runs a debugger script from file.

Usage
```
run <"fileName">
```

Parameters

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

s

Purpose Single steps the processor, stepping into subroutines.

Usage
```
s
```

Parameters None.

Example
```
s

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

save

Purpose Saves a range of data from memory to file.

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

Parameters

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

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

sb

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

Usage
```
sb <addr> <value>
```

Parameters

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

Example
```
sb 0111 0a

Memory set starting at 00000111
```

sc

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

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

Parameters

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

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

Example

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

sc6

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

Usage

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

Parameters

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

Comments This command is the same as the sc command.

Example

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

sc7

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

Usage

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

Parameters

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

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

Example

```
sc7

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

sizeof

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

Usage

```
sizeof <template>
```

Parameters

`template`	The name of the template.

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

Example

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

sl

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

Usage

```
sl <addr> <value>
```

Parameters

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

Example

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

SS

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

Usage

```
ss [<addr>]
```

Parameters

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

Example

```
ss 1000F024
```

storeinfo

Purpose Displays information about a memory store.

Usage

```
storeinfo <cardNum>
```

Parameters

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

Comments You can use the `storeinfo` command in either the Console window or the debugging window.

Example

```
storeinfo 0

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

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

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

sw

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

Usage
```
sw <addr> <value>
```

Parameters

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

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

t

Purpose Single steps the processor, stepping over subroutines.

Usage
```
t
```

Parameters None.

Example
```
t

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

templates

Purpose Lists the names of the debugger templates.

Usage
```
templates
```

Parameters None.

Example
```
templates

Char
Byte
SByte
Word
SWord
```

```
DWord
SDWord
```

typedef

Purpose Begins a structure definition block.

Usage
```
typedef struct <"name">
```

Parameters

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

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

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

typeend

Purpose Ends a structure definition block.

Usage
```
typeend
```

Parameters None.

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

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

var

Purpose Defines a debugger variable.

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

Parameters

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

Example
```
var "testvar" 100

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

variables

Purpose Lists the names of the debugger variables.

Usage
```
variables
```

Parameters None.

Example
```
variables

DebOut
SymbolsOn
ReadMemHack
StepRegs
Attached
testvar
testvar2
```

wh

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

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

Parameters

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

`ATrapNumber`	The number of the A-trap number for which you want information displayed.

Debugging Command Summary

Flow Control Commands

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

Memory Commands

atr	Registers a function name with an A-Trap number.
db	Displays the byte value at a specified address.
dl	Displays the 32-bit long value at a specified address.
dm	Displays memory for a specified number of bytes or templates.
dw	Displays the 16-bit word value at a specified address.
fb	Searches through a range of memory for a specified byte value.
fill	Fills memory with a specified byte value.
fl	Searches through a range of memory for a specified 32-bit long value.
ft	Searches through a range of memory for the specified text.
fw	Searches through a range of memory for the specified 16-bit word value.
il	Disassembles code in a specified line range.
sb	Sets the value of the byte at the specified address.
sc	Lists the A6 stack frame chain, starting at the specified address.
sc6	Lists the A6 stack frame chain, starting at the specified address.
sc7	Lists the A7 stack frame chain, starting at the specified address.
sl	Sets the value of the long at the specified address.

sw	Sets the value of the word at the specified address.
wh	Displays system function information for a specified function name or A-Trap number. Also identifies the memory chunk that contains a specific address or lists all system functions.

Template Commands

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

Register Commands

reg	Displays all registers.

Utility Commands

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

Console Commands

cardinfo	Retrieves information about a memory card.
dir	Lists the databases.
dump	Dumps a range of memory to a file.
hchk	Checks a heap.
hd	Displays a dump of a memory heap.
hl	Lists all of the memory heaps on the specified memory card.
ht	Performs a heap total.
info	Displays information on a heap chunk.
opened	Lists all currently opened databases.
storeinfo	Retrieves information about a memory store.

Miscellaneous Debugger Commands

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

Debugger Environment Variables

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

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

Predefined Constants

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

3

Debugger Protocol Reference

This chapter 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 from 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.

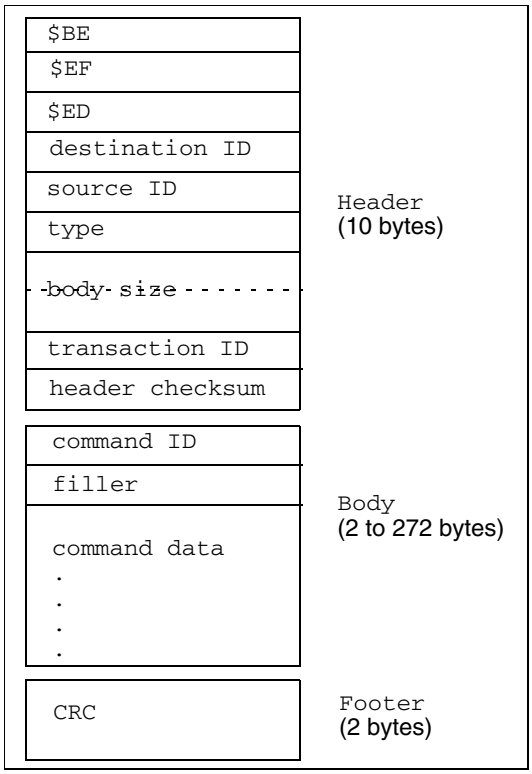
Although the typical flow of packets involves the host sending a request and the target sending back a response, although there are a some exceptions, as follows:

- 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 91 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 sysPktMaxMemChunk   256
#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 dbgNormalBreakpoints  5
#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	sysPktGetTrap ConditionalsCmd	sysPktGetTrap ConditionalsRsp
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	sysPktSetTrap ConditionalsCmd	sysPktSetTrap ConditionalsRsp
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.

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 89.

Commands

The `Get Breakpoints` command request and response commands are defined as follows:

```
#define sysPktGetBreakpointsCmd 0x0B
#define sysPktGetBreakpointsRsp 0x8B
```

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 sysPktGetRtnNameCmd 0x04
#define sysPktGetRtnNameRsp 0x84
```

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;
     charname[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 sysPktGetTrapBreaksCmd 0x10
#define sysPktGetTrapBreaksRsp 0x90
```

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 sysPktGetTrapConditionsCmd 0x14
#define sysPktGetTrapConditionsRsp 0x94
```

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 sysPktRemoteMsgCmd  0x7F
```

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 sysPktReadMemCmd 0x01
#define sysPktReadMemRsp 0x81
```

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 sysPktReadRegsCmd 0x05
#define sysPktReadRegsRsp 0x85
```

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.

Commands The `RPC` request and response commands are defined as follows:

```
#define sysPktRPCCmd 0x0A
#define sysPktRPCRsp 0x8A
```

Request Packet

```
typedef struct SysPktRPCType
{
     _sysPktBodyCommon;
     Word trapWord;
     DWord resultD0;
     DWord resultD0;
     Word numParams;
     SysPktRPCParamType param[?];
}
```

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 sysPktSetBreakpointsCmd 0x0C
#define sysPktSetBreakpointsRsp 0x8C
```

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 sysPktSetTrapBreaksCmd 0x0C
#define sysPktSetTrapBreaksRsp 0x8C
```

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 sysPktSetTrapConditionsCmd  0x15
#define sysPktSetTrapConditionsRsp  0x95
```

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 sysPktStateCmd 0x00
#define sysPktStateRsp 0x80
```

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];
     BreakpointType bp[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 sysPktDbgBreakToggleCmd 0x0D
#define sysPktDbgBreakToggleRsp 0x8D
```

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 sysPktWriteMemCmd 0x02
#define sysPktWriteMemRsp 0x82
```

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 sysPktWriteRegsCmd 0x06
#define sysPktWriteRegsRsp 0x86
```

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.

Table 3.2 Debugger protocol command packets *(continued)*

Command	Description
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.
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.

4

Using the Console Window

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

The following topics are covered in this chapter:

About the Console Window

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

The console is available in three environments:

- as a separate window for sending and receiving commands in the Palm Debugger program, which is described in Chapter 1, "Using Palm Debugger."
- as a separate window that you can open from within Palm Simulator program.
- as a separate window that you can open within the Metrowerks CodeWarrior environment.

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

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

To learn more about using console commands, see the section "Entering Console Window Commands" on page 117. For a complete reference description of each console command, see "Console Window Commands" on page 122. The commands are summarized in "Console Command Summary" on page 155.

Connecting the Console Window

Activating Console Input

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

The console nub runs as a background thread on the device, listening for commands on the serial or USB port. To activate the console nub, use the , as described in "Using Shortcut Numbers to Activate the Windows" on page 115.

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

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

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

Verifying Your Connection

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

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

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

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

Using Shortcut Numbers to Activate the Windows

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

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

To enter a shortcut number, follow these steps:

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

2. Next, tap the stylus twice, to generate a dot (a period).
3. Next, draw a number character in the number entry portion of the device's text entry area. Table 4.1 shows the different shortcut numbers that you can use.

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

Table 4.1 Shortcut Numbers for Debugging

Number	Description	Notes
	The device enters debugger mode, and waits for a low-level debugger to connect. A flashing square appears in the top left corner of the device.	This mode opens a serial port, which drains power over time. You must perform a soft reset or use the debugger's `reset` command to exit this mode.

Table 4.1 Shortcut Numbers for Debugging *(continued)*

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

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

Entering Console Window Commands

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

NOTE: Console command input is not case sensitive.

Table 4.2 shows the most commonly used console window commands.

Table 4.2 Commonly Used Console Commands

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

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

Listing 4.1 Importing a Database into the Handheld Device

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

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

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

dir 0
name                    ID      total       data
-------------------------------------------------
```

```
*System             00D20A44  392.691 Kb  390.361 Kb
*AMX                00D209C4   20.275 Kb   20.123 Kb
*UIAppShell         00D20944    1.327 Kb    1.175 Kb
*PADHTAL Library    00D208E2    7.772 Kb    7.674 Kb
*IrDA Library       00D20876   39.518 Kb   39.402 Kb
*Net Library        00D207E2   86.968 Kb   86.780 Kb
*PPP NetIF          00D2073A   30.462 Kb   30.238 Kb
*SLIP NetIF         00D20692   15.812 Kb   15.588 Kb
*Loopback NetIF     00D20630    1.810 Kb    1.712 Kb
*MS-CHAP Support    00D205C4    4.342 Kb    4.226 Kb
*Network            00D203D2   40.442 Kb   39.624 Kb
*Address Book       00D20226   59.825 Kb   59.133 Kb
*Calculator         00D2002A   14.597 Kb   13.761 Kb
*Date Book          00D1FCF8  106.200 Kb  104.806 Kb
*Launcher           00D1FA98   36.633 Kb   35.617 Kb
*Memo Pad           00D1F91E   24.267 Kb   23.665 Kb
*Preferences        00D1F876    1.403 Kb    1.179 Kb
*Security           00D1F706    8.414 Kb    7.830 Kb
*HotSync            00D1F334   39.078 Kb   37.396 Kb
*To Do List         00D1F1E2   33.232 Kb   32.702 Kb
*Digitizer          00D1F126    2.002 Kb    1.742 Kb
*General            00D1EFE8    8.749 Kb    8.255 Kb
*Formats            00D1EF4A    4.732 Kb    4.526 Kb
*ShortCuts          00D1EE34    6.499 Kb    6.077 Kb
*Owner              00D1ED5A    4.095 Kb    3.781 Kb
*Buttons            00D1EC4E    7.419 Kb    7.015 Kb
*Modem              00D1EB74    8.222 Kb    7.908 Kb
*Mail               00D1E838   59.765 Kb   58.353 Kb
*Expense            00D1E614   42.304 Kb   41.396 Kb
*Unsaved Preferences  0001811B    0.898 Kb    0.550 Kb
*Net Prefs          00018133    0.084 Kb    0.000 Kb
 AddressDB          00018137   66.149 Kb   51.945 Kb
 MemoDB             0001815F    2.186 Kb    1.902 Kb
 ToDoDB             00018173    1.000 Kb    0.876 Kb
 MailDB             0001817F    1.033 Kb    0.929 Kb
 DatebookDB         000181EB   53.162 Kb   29.678 Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842 Kb
*Saved Preferences  00018123    3.753 Kb    3.031 Kb
 NetworkDB          0001818B    0.986 Kb    0.722 Kb
*Giraffe High Score  00018273    0.126 Kb    0.020 Kb
 Datebk3DB          0001827B    0.084 Kb    0.000 Kb
 ReDoDB             0001827F    0.084 Kb    0.000 Kb
 LauncherDB         0001814F    0.294 Kb    0.190 Kb
*MineHunt           00018287    9.810 Kb    9.264 Kb
*SubHunt            000182DF   17.700 Kb   16.758 Kb
*Puzzle             0001837F    5.256 Kb    4.886 Kb
*HardBall           000183B7   18.877 Kb   18.177 Kb
 Pictures           0001842B    0.084 Kb    0.000 Kb
```

```
*Jot                0001842F  120.409 Kb  119.841 Kb
*Graffiti ShortCuts    001FFE7F    2.872 Kb    2.766 Kb
*UnDupe             001FFE87    9.462 Kb    9.070 Kb
*WordView           001FFEC3   17.320 Kb   16.752 Kb
*SheetView          001FFF1F   56.753 Kb   55.877 Kb
 AOU Birds of NA    001FFE15  130.265 Kb   90.021 Kb
 ExpenseDB          001FBCB5    0.150 Kb    0.046 Kb
 DocsToGoDB         001FBCC1    0.326 Kb    0.202 Kb
 birds.PDB          001FBCD1    0.709 Kb    0.585 Kb
 foo                0001812F    0.084 Kb    0.000 Kb
*Text To Hex        001FFF85   34.725 Kb   33.827 Kb
------------------------------------------------------------
---
Total: 59
```

These and all of the other console commands are described in detail in "Console Window Commands" on page 122.

Command Syntax

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

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

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

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

The following is an example of a command definition

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

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

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

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

Specifying Command Options

All command options and some command parameters are specified as flags that begin with a dash. For example:

```
-c
-enable
```

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

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

NOTE: You use the dash (-) character to specify options for console commands. If you are using Palm Debugger, you must use the backslash (\) character to specify options for commands that you type in the debugging window; this is because the expression parser used for debugging commands interprets the dash as a minus sign.

Specifying Numeric and Address Values

Many of the console commands take address or numeric arguments. You can specify these values in hexadecimal, decimal, or binary. All values are assumed to be hexadecimal unless preceded by a sign that specifies decimal (`#`) or binary (`%`). Table 4.3 shows values specified as binary, decimal, and hexadecimal in a debugging command:

Table 4.3 Specifying Numeric Values in Palm Debugger

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

Console Window Commands

You use the console window to send commands to the console nub that is running on the handheld device.

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

Table 4.4 Console Window Command Categories

Command category	Commands
Card Information	cardformat, cardinfo, and storeinfo.
Chunk Utility	free, info, lock, new, resize, setowner, and unlock.
Database Utility	close, create, del, dir, export, import, open, opened, and setinfo.
Debugging Utility	dm, gdb, mdebug, and sb.
Gremlin	gremlin and gremlinoff.
Heap Utility	hc, hchk, hd, hf, hi, hl, hs, ht, and htorture.
Host Control	help, log, and saveimages.
Miscellaneous Utility	simsync and sysalarmdump.
Record Utility	addrecord, delrecord, detachrecord, findrecord, listrecords, moverecord, and setrecordinfo.

Table 4.4 Console Window Command Categories *(continued)*

Command category	Commands
Resource Utility	addresource, attachresource, changeresource, delresource, detachresource, listresources,and setresourceinfo.
System	battery, coldboot, doze, exit, feature, kinfo, launch, performance, poweron, reset, sleep, and switch.

addrecord

Purpose Adds a record to a database.

Usage

```
addrecord <accessPtr> <index> <recordText>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.
`recordText`	The record data.

addresource

Purpose Adds a resource to a database.

Usage

```
addresource <accessPtr> -t <type> -id <id>
   <resourceText>
```

Parameters

`accessPtr`	A pointer to the database.
`type`	The type of the resource that you are adding.
`id`	The ID for the resource that you are adding.
`resourceText`	The resource data.

attachrecord

Purpose Attaches a record to a database.

Usage

```
attachrecord <accessPtr> <recordHandle> <index>
    [options]
```

Parameters

`accessPtr` A pointer to the database.

`recordHandle` A handle to the record that you are attaching to the database.

`index` The index of the record.

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

`-r` Replaces the existing record with the same index, if one exists.

attachresource

Purpose Attaches a resource to a database.

Usage

```
attachrecord <accessPtr> <recordHandle> <index>
    [options]
```

Parameters

`accessPtr` A pointer to the database.

`recordHandle` A handle to the resource that you are attaching to the database.

`index` The index of the resource.

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

`-r` Replaces the existing resource with the same index, if one exists.

battery

Purpose A battery utility command for performing battery operations.

Usage

```
battery [options]
```

Parameters

`options` Optional. Specifies the battery operation to perform. Use one of the following values:

`-rStart <deltaSeconds>`
: Start radio charging in the number of seconds specified by `deltaSeconds`.

`-rStop`
: Stop radio charging.

`-rLoaded` (`yes` | `no`)
: Set loaded state to `yes` or `no`.

Example `battery -rStop`

cardformat

Purpose Formats a memory card.

Usage
```
cardformat <cardNum> <cardName> <manufName>
    <ramStoreName>
```

Parameters

`cardNum`	The card number.
`cardName`	The name to associate with the card.
`manufName`	The manufacturer name to associate with the card.
`ramStoreName`	The RAM store name to associate with the card.

cardinfo

Purpose Displays information about a memory card.

Usage
```
cardinfo <cardNum>
```

Parameters

`cardNum`	The card number about which you want information. You can use `0` to specify the built-in RAM.

Example
```
cardinfo 0

Name: PalmCard
Manuf: Palm, Inc
Version: 0001
CreationDate: B1243780
ROM Size: 00118FFC
RAM Size: 00200000
```

```
Free Bytes : 0015ACB2
Number of heaps: #3
```

changerecord

Purpose Replaces a record in a database.

Usage
```
changerecord <accessPtr> <index> <recordText>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.
`recordText`	The new record data.

changeresource

Purpose Replaces a resource in a database.

Usage
```
changeresource <accessPtr> <index> <recordText>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.
`resourceText`	The new resource data.

close

Purpose Closes a database.

Usage
```
close <accessPtr>
```

Parameters

`accessPtr`	A pointer to the database.

coldboot

Purpose Initiates a hard reset on the handheld device.

Usage
```
coldboot
```

Parameters None

Comments Use the `coldboot` command to perform a hard reset of the handheld device. A hard reset erases all data on the device, restoring it to its new condition.

The handheld device requires confirmation of this operation. You are prompted to press the Up button on the device to confirm that you want to perform a hard reset, or press any other button to cancel the operation.

Example

```
coldboot
```

create

Purpose Creates a new database on the handheld device.

Usage

```
create <cardNum> <name> [options]
```

Parameters

`cardNum` The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.

`name` The name for the new database on the handheld device.

`options` Optional. Specifies information about the new database:

`-t <type>`
The 4-character database type identifier.

`-c <creator>`
The 4-character database creator ID.

`-v <version>`
The database version number.

`-r` Specify to indicate that the database is a resource database.

Comments Use the create command to create a new record or resource database on the handheld device.

del

Purpose Deletes a database from the handheld device.

Usage

```
del <cardNum> <fileName>
```

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
`fileName`	The name of the database on the handheld device. Note that you must quote the database name if it contains spaces.

Comments Use the `del` command to delete a database from the specified card on the handheld device.

You can get a list of the databases on the device with the dir command.

You cannot delete an open database.

Result If the database you want to delete is not found or is currently opened, you receive an error message.

Example

```
del 0 birds.pdb

Success!!
```

delrecord

Purpose Deletes a record from a database.

Usage

```
delrecord <accessPtr> <index>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.

Comments Use the `delrecord` command to delete the record at the specified `index` value from the database specified by `accessPtr`.

delresource

Purpose Deletes a resource from a database.

Usage
```
delresource <accessPtr> <index>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.

Comments Use the `delresource` command to delete the resource at the specified `index` value from the database specified by `accessPtr`.

detachrecord

Purpose Detaches a record from a database.

Usage
```
detachrecord <accessPtr> <index>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the record in the database.

Comments Use the `detachrecord` command to detach the record at the specified `index` value from the database specified by `accessPtr`.

detachresource

Purpose Detaches a resource from a database.

Usage
```
detachresource <accessPtr> <index>
```

Parameters

`accessPtr`	A pointer to the database.
`index`	The index of the resource in the database.

Comments Use the `detachresource` command to detach the resource at the specified `index` value from the database specified by `accessPtr`.

dir

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

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

Parameters

`cardNum`	The card number whose databases you want listed. You almost always use `0` to specify the built-in RAM.

`searchOptions` Optional. Options for listing a specific database. Specify any combination of the following flags.

`-c <creatorID>`
Search for a database by creator ID.

`-latest`
List only the latest version of each database.

`-t <typeID>`
Search for a database by its type.

`displayOptions` Optional. Options for which information is displayed in the listing. Specify any combination of the following flags.

`-a`	Show all information.
`-at`	Show the database attributes.
`-d`	Show the database creation, modification, and backup dates.
`-i`	Show the database appInfo and sortInfo field values.
`-id`	Show the database chunk ID
`-s`	Show the database size
`-m`	Show the database modification number.
`-n`	Show the database name.
`-r`	Show the number of records in the database.
`-tc`	Show the database type ID and creator ID.
`-v`	Show the database version number.

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

```
dir 0
```

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

```
dir 0 -a
```

Example

```
dir 0

name                      ID      total       data
-------------------------------------------------
*System             00D20A44  392.691 Kb  390.361 Kb
*AMX                00D209C4   20.275 Kb   20.123 Kb
*UIAppShell         00D20944    1.327 Kb    1.175 Kb
*PADHTAL Library    00D208E2    7.772 Kb    7.674 Kb
*IrDA Library       00D20876   39.518 Kb   39.402 Kb
 ...
 MailDB             0001817F    1.033 Kb    0.929 Kb
 NetworkDB          0001818B    0.986 Kb    0.722 Kb
 System MIDI Sounds  000181B3    1.066 Kb    0.842 Kb
 DatebookDB         000181FB    0.084 Kb    0.000 Kb
-------------------------------------------------
Total: 41
```

dm

Purpose Displays a range of memory values.

Usage

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

Parameters

`addr`	The starting memory address to be displayed.
`count`	The number of bytes to be displayed. If this is omitted, eight bytes of data are displayed.

Example

```
dm 0000f000

0000F000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
```

doze

Purpose Instructs the handheld device's CPU to sleep while maintaining the peripherals and the clock.

Usage

```
doze [options]
```

Parameters

`options`	You can optionally specify the following flags:

`-light`
The handheld device will awaken in response to any interrupt.

Example `doze -light`

exit

Purpose Exits the debugger.

Usage `exit`

Parameters None.

export

Purpose Copies a Palm OS database from the handheld device to the desktop computer.

Usage `export <cardNum> <fileName>`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
`fileName`	The name of the database on the handheld device. Note that you must quote the database name if it contains spaces.

Comments Use the `export` command to copy a database from the handheld device to your desktop computer. You can get a list of the databases on the device with the dir command.

If the database contains resources, it is copied in standard PRC format; if the database contains records, it is copied in standard PDB format. Note that these two formats are actually identical.

The exported file is stored in the `Device` subdirectory of the directory in which Palm Debugger executable is stored.

The exported file is named `fileName`, with no added extensions.

Example

```
export 0 "Text to Hex"

Exporting resource 'code'=0....
Exporting resource 'data'=0....
Exporting resource 'pref'=0....
```

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

feature

Purpose Accesses features.

Usage `feature [options]`

Parameters `options` Optional. You can use the following options:

`-all`
Displays a list of all known features

`–unreg <creator> <num>`
Unregisters the specified feature

`-get <creator> <num>`
Displays the value of a feature

`-set <creator> <num> <value>`
Sets the value of a feature.

Example

```
feature -all

ROM:  creator   number   value
      'psys'     #1      03003000
      'psys'     #2      00010000
RAM:  creator   number   value
      'psys'     #3      00000001
      'psys'     #4      00000001
      'psys'     #7      00000001
      'netl'     #0      02003000
      'irda'     #0      03003000

feature -get psys 3

Value = 00000001
```

findrecord

Purpose Finds a record by ID.

Usage `findrecord <accessPtr> <id>`

Parameters

`accessPtr` A pointer to the database.

`id` The unique record ID.

free

Purpose Disposes of a chunk.

Usage

```
free (<hexChunkPtr> | localID>) [options]
```

Parameters

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

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

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

gdb

Purpose Enables or disables gdb debugging

Usage

```
gdb [options]
```

Parameters

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

`-enable`
Enables gdb debugging.

`-disable`
Disables gdb debugging.

getresource

Purpose Retrieves the specified resource.

Usage

```
getresource -t <type> -id <id>
```

Parameters

`type` The type of resource that you want to retrieve.

`id` The ID of the resource that you want to retrieve.

gremlin

Purpose Activates a Gremlin until the specified event occurs.

Usage
```
gremlin <num> <until>
```

Parameters

`num`	The number of the Gremlin to activate.
`until`	The event that deactivates the Gremlin.

gremlinoff

Purpose Deactivates the current Gremlin.

Usage
```
gremlinoff
```

Parameters None

Example
```
gremlinoff
```

hc

Purpose Compacts a memory heap.

Usage
```
hc <heapId>
```

Parameters

`heapId`	The hexadecimal number of the heap to be compacted. Heap number `0x0000` is always the dynamic heap.

Example
```
hc 0002
Heap Compacted
```

hchk

Purpose Checks the integrity of a heap.

Usage
```
hchk <heapId> [options]
```

Parameters

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

Example
```
hchk 0000
Heap OK
```

hd

Purpose Displays a hexadecimal dump of the specified heap.

Usage

```
hd <heapId>
```

Parameters

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

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

Example

```
hd 0

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

...

000114E8 -------- F00114E8 000FF8 001008  #0  #0    FM
-000124F0 -------- F00124F0 001000 001008  #0 #15    fM
-00017D30 -------- F0017D30 00003C 000044  #0 #15    fM
SysAppInfoPtr: AMX
-00017D74 -------- F0017D74 000008 000010  #0 #15    fM
Feature Manager Globals (FtrGlobalsType)
-00017D84 -------- F0017D84 000024 00002C  #0 #15    fM
DmOpenInfoPtr: 'Update 3.0.2'
-00017DB0 -------- F0017DB0 00000E 000016  #0 #15    fM
DmOpenRef: 'Update 3.0.2'
```

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

help

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

Usage

```
help
    help <command>
```

Parameters

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

Example

```
help hchk

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

hf

Purpose Allocates almost all of the free bytes in a heap, reserving the specified amount of free space.

Usage

```
hf <heapId> <freeBytes>
```

Parameters

`heapId`	The hexadecimal number of the heap. Heap number `0x0000` is always the dynamic heap.
`freeBytes`	The number of bytes to leave unallocated.

Example

```
hf 0000 20
```

hi

Purpose Initializes the specified memory heap.

Usage
```
hi <heapId>
```

Parameters

`heapId`	The hexadecimal number of the heap to be initialized. Heap number `0x0000` is always the dynamic heap.

Example
```
hi 0006
```

hl

Purpose Displays a list of memory heaps.

Usage
```
hl <cardNum>
```

Parameters

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

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

Example
```
hl 0
```

```
   index   heapID  heapPtr     size     free  maxFree    flags
-------------------------------------------------------------
       0     0000 00001480 00016B80 00010C50 0000EC48     8000
       1     0001 1001810E 001E7EF2 0014AD6A 00147D3A     8000
       2     0002 10C08212 00118DEE 0000A01C 0000A014     8001
```

hs

Purpose Scrambles the specified heap.

Usage
```
hs <heapId>
```

Parameters

`heapId`	The hexadecimal number of the heap to be scrambled. Heap number `0x0000` is always the dynamic heap.

Comments Scrambling a heap moves its contents around. You can use this to verify that the program is using handles in the prescribed manner.

Example
```
hs 0002
heap scrambled
```

ht

Purpose Displays summary information for the specified heap.

Usage
```
ht <heapId>
```

Parameters

`heapId` The hexadecimal number of the heap to be scrambled. Heap number `0x0000` is always the dynamic heap.

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

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

htorture

Purpose Tortures a heap to test its integrity.

Usage
```
htorture <heapId> [options]
```

Parameters

`heapId` The hexadecimal number of the heap to be tortured. Heap number `0x0000` is always the dynamic heap.

`options` Optional. You can specify a combination of the following options:

`-c` Checks the contents of every chunk.

`-f <number>`
Reports if the heap is filled beyond the specified percentage. The default is `90` percent.

`-l <filename>`
Specifies the name of the log file

`-m <hexSize>`
: The maximum chunk size. The default value is `0x400`.

`-p <level>`
: The progress level to display. Specify a number between `0` (minimum detail) and `2` (maximum detail). The default value is `0`.

Comments Use the `htorture` command to torture-test a memory heap. You can specify a logging file to which the output of the test is sent. You can also use the `-p` command to control how progress is displayed.

import

Purpose Copies a Palm OS database from the desktop computer to the handheld device.

Usage

```
import <cardNum> <fileName>
```

Parameters

`cardNum` The card number on which the database is to be installed. You almost always use `0` to specify the built-in RAM.

`fileName` The name of the file on the desktop computer. You can specify an absolute file name path, or a relative file name path.

The default search path is the `Device` subdirectory of the directory in which Palm Debugger executable is stored.

Comments Use the `import` command to load a new version of your application or database onto the handheld device.

This command provides a more convenient install operation and has the same functionality as the installer tool provided with the HotSync Manager application.

The name of the database on the handheld device is the name stored in the file, and is not the same as the file name. If a database with a matching name is already open on the handheld device, an error is generated. If a database with a matching name is already stored on

the handheld device, that database is deleted and replaced by the file.

Result If a database with a matching name is currently open on the handheld device, the `dmErrAlreadyExists` error code (`0x0219`) is generated.

Example

```
import 0 Tex2HexApp.prc

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

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

info

Purpose Displays information about a memory chunk.

Usage

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

Parameters

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

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

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

kinfo Purpose

Displays a list of all system kernel information.

Usage

```
kinfo [options]
```

Parameters

`options` Optional. Specify the kernel information that you want to see displayed. Use a combination of the following flags:

`-all`
Display all kernel information.

`-task (<id> | all)`
Display task information.

`-sem (<id> | all)`
Display semaphore information.

`-tmr (<id> | all)`
Display timer information.

Comments

Use the `kinfo` command to display a list of system kernel information, including tasks, semaphores, event groups, and timers.

Example

```
kinfo -all
```

```
Task Information:
  taskID     tag  priority  stackPtr   status
  ------------------------------------------------------------------
  000176EA    AMX  #   0    00017556   Idle: Waiting for Trigger
  000178BE   psys  #  30    00013364   Waiting on event timer
  0001795A   CONS  #  10    0001103E   Running

Semaphore Information:
  semID      tag   type      initValue  curValue        nesting    ownerID
  ------------------------------------------------------------------------
  000177EE   MemM  resource   #-1        #1  (free)      #0        00000000
  00017822   SlkM  counting   #1         #1  (avail.)    #0        00000000
  0001788A   SndM  counting   #1         #1  (avail.)    #0        00000000
  00017A5E   SerM  counting   #0         #0  (unavail.)  #0        00000000

Timer Information:
  tmrID      tag   ticksLeft   period   procPtr
  ------------------------------------------
  000177BA   psys    #  83    #   0    10C6C618
```

launch

Purpose Launches an application on the handheld device.

Usage
```
launch [-t] [-ns] [-ng] <cardNum> <name> [<cmd>
    <cmdStr>
```

Parameters

`-t`	Launches the application as a separate task.
`-ns`	Use the caller's stack.
`-ng`	Use the caller's globals environment.
`cardNum`	The card number on which application is located. You almost always use `0` to specify the built-in RAM.
`name`	The name of the application to be launched.
`cmd`	Optional. Use to specify a command for the application.
`cmdStr`	Optional. Use to specify an arguments string for `cmd`.

listrecords

Purpose Lists the records in a database.

Usage
```
listrecords <accessPtr>
```

Parameters

`accessPtr`	A pointer to the database.

listresources

Purpose Lists the resources in a database.

Usage
```
listresources <accessPtr>
```

Parameters

`accessPtr`	A pointer to the database.

lock

Purpose Locks a memory chunk.

Usage

```
lock (<hexChunkPtr> | localID>) [options]
```

Parameters

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

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

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

log

Purpose Toggles logging of debugger output to a file.

Usage

```
log <fileName>
```

Parameters

`fileName` The name of the file to which debugger output is sent.

Comments Use the `log` command to start or stop logging of debugger output to a file.

mdebug

Purpose Sets the Memory Manager debug mode, which you can use to track down memory corruption problems.

Usage

```
mdebug [options]
```

Parameters

`options` Optional. Specify the kernel information that you want to see displayed. Use a combination of the following flags:

`-full`
Shortcut for full debugging.

`-partial`
Shortcut for partial debugging.

`-off`
Shortcut to disable debugging.

`-a` Check/scramble all heaps each time.

`-a-` Check only the heap currently in use.

`-c` Check heap(s) on some memory calls.

`-ca` Check heap(s) on all memory calls.

`-c-` Do not check heaps.

`-f` Check free chunk contents.

`-f-` Do not check free chunk contents.

`-min`
Store minimum available free space in dynamic heap in the global variable `GMemMinDynHeapFree`.

`-min-`
Do not record minimum free space.

`-s` Scramble heap(s) on some memory calls.

`-sa` Scramble heap(s) on all memory calls.

`-s-` Do not scramble heaps.

Comments Use the `mdebug` command to enable debugging for tracking down memory corruption problems.

IMPORTANT: The different debug modes enabled by `mdebug` can significantly slow down operations on the handheld device. Full checking is slowest, partial checking is slow, and only enabling specific options is the fastest.

Example

```
mdebug -full
Current mode = 003A
  Every heap checked/scrambled per call
  Heap(s) checked on EVERY Mem call
  Heap(s) scrambled on EVERY Mem call
  Free chunk contents filled & checked
   Minimum dynamic heap free space recording OFF
```

moverecord

Purpose Moves a record in the database by changing its index.

Usage

```
moverecord <accessPtr> <fromIndex> <toIndex>
```

Parameters

`accessPtr` A pointer to the database.

`fromIndex` The original index of the record in the database.

`toIndex` The new index for the record in the database.

new

Purpose Allocates a new chunk in a heap.

Usage

```
new <heapId> <hexChunkSize> [options]
```

Parameters

`heapId` The hexadecimal number of the heap in which to allocate a new chunk. Heap number `0x0000` is always the dynamic heap. Note that `heapId` is ignored if you specify the `-near` option.

`hexChunkSize` The number of bytes in the new chunk, specified as a hexadecimal number.

`options` Optional. You can specify a combination of the following options:

`-c` Fill the chunk contents.

`-lock`
Pre-lock the chunk.

`-n` Make the chunk unmoveable.

`-near <ptr>`
Allocate the new chunk in the same heap as the specified pointer. If this option is specified, the `heapId` is ignored.

`-o <ownerId>`
Set the owner of the chunk to the specified ID value.

open

Purpose Opens a database.

Usage `open <cardNum> <name> [options]`

Parameters

`cardNum`	The card number on which the database is located. You almost always use `0` to specify the built-in RAM.
`name`	The name of the database.
`options`	Optional. You can specify the following options: `-r` Open the database for read-only access. `-p` Leave the database open.

opened

Purpose Lists all of the currently opened databases.

Usage `opened`

Parameters None.

Example `opened`

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

performance

Purpose Sets the performance level of the handheld device.

Usage `performance [options]`

Parameters

`options`	You can specify the following options: `-b <baud>` Uses the specified <baud> rate to

calculate the nearest clock frequency value.

`-d <duty>`
Set the CPU duty cycle. The <duty> value specifies the number of CPU cycles out of every 31 system clock ticks.

`-f <freq>`
Set the system clock frequency to the specified Hz value; select the nearest baud multiple as the frequency.

`-ff <freq>`
Set the system clock frequency to the specified Hz value; do not pick the nearest baud multiple.

poweron

Purpose Powers on the handheld device.

Usage
```
poweron
```

Parameters None.

Example
```
poweron
```

reset

Purpose Performs a soft reset on the handheld device.

Usage
```
reset
```

Parameters None.

Comments This command performs the same reset that is performed when you press the recessed reset button on a Palm Powered handheld device.

Example
```
reset
Resetting system
```

resize

Purpose Resizes an existing memory chunk.

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

Parameters

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

`hexNewSize` The new size of the chunk, in bytes.

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

`-c` Checks and fills the contents of the resized chunk.

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

saveimage

s

Purpose Saves a memory card image.

Usage
```
saveimages
```

Parameters None.

sb

Purpose Sets the value of a byte in memory.

Usage
```
sb <addr> <value>
```

Parameters

`addr` The address of the byte.

`value` The new value of the byte.

setinfo

Purpose Sets new information values for a database.

Usage

```
setinfo <cardNum> <dbName> [options]
```

Parameters

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

dbName — The name of the database.

options — Options. You can specify a combination of the following values:

–m <modification>
: Sets the modification number for the database.

–n <name>
: Sets the name of the database.

–v <version>
: Sets the version number of the database.

setowner

Purpose Sets the owner ID of a memory chunk.

Usage

```
setowner (<hexChunkPtr> | <localID>) <owner>
    [options]
```

Parameters

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

`hexNewSize` — The new size of the chunk, in bytes.

`owner` — The new owner ID for the chunk.

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

`-card <cardNum>`
: The card number if a local ID is specified instead of a chunk pointer. Use `0` to specify the built-in RAM.

setrecordinfo

Purpose Changes information for a record in a database.

Usage
```
setrecordinfo <accessPtr> <index> [options]
```

Parameters

`accessPtr` A pointer to the database.

`index` The index of the record in the database.

`options` Optional. You can specify a combination of the following options:

`-a <hexAttr>`
Sets attribute bit settings for the record.

`-u <uniqueId>`
Sets unique record ID for the record.

setresourceinfo

Purpose Changes information for a resource in a database.

Usage
```
setresourceinfo <accessPtr> <index> [options]
```

Parameters

`accessPtr` A pointer to the database.

`index` The index of the resource in the database.

`options` Optional. You can specify a combination of the following options:

`-t <resType>`
Sets resource type for the resource.

`-id <resId>`
Sets resource ID for the resource.

simsync

Purpose Simulates a synchronization operation on a specific database.

Usage
```
simsync <accessPtr>
```

Parameters

`accessPtr` A pointer to the database.

sleep

Purpose Shuts down all peripherals, the CPU, and the system clock.

Usage

```
sleep
```

Parameters None.

storeinfo

Purpose Displays information about a memory store.

Usage

```
storeinfo <cardNum>
```

Parameters

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

Example

```
storeinfo 0

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

RAM Store:
  version: 0001
  flags: 0001
  name: RAM Store 0
  creation date: 00000000
  backup date: 00000000
  heap list offset: 00018100
  init code offset1: 00000000
  init code offset2: 00000000
  database dirID: 0001811F
```

switch

Purpose Switches the application that is used to provide the user interface on the handheld device.

Usage
```
switch <cardNum> <name> [<cmd> <cmdStr>]
```

Parameters

`cardNum`	The number of the card on which the user interface application is stored. You almost always use `0` to specify the built-in RAM.
`name`	The name of the application.
`cmd`	Optional. Use to specify a command for the application.
`cmdStr`	Optional. Use to specify an arguments string for `cmd`.

sysalarmdump

Purpose Displays the system alarm table.

Usage
```
sysalarmdump
```

Parameters None.

Example `sysalarmdump`

```
                          alarm            card
   date    time     ref    seconds     dbID    #     quiet  triged noted
-------------------------------------------------------------------------
 7/29/1999  00:00  00000000  B3C54A00  00D1FCF8  4004  false  false  false
 1/ 1/1904  00:00  00000000  00000000  00000000  0000  false  false   true
```

unlock

Purpose Unlocks a memory chunk.

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

Parameters

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

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

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

Console Command Summary

Card Information Commands

cardformat	Formats a memory card.
cardinfo	Retrieves information about a memory card.
storeinfo	Retrieves information about a memory store.

Chunk Utility Commands

free	Disposes of a heap chunk.
info	Displays information on a heap chunk.
lock	Locks a heap chunk.
new	Allocates a new chunk in a heap.
resize	Resizes an existing heap chunk.
setowner	Sets the owner of a heap chunk.
unlock	Unlocks a heap chunk.

Database Utility Commands

close	Closes a database.
create	Creates a new database.
del	Deletes a database.
dir	Lists the databases.
export	Exports a database to the desktop computer.
import	Imports a database from the desktop computer.
open	Opens a database.
opened	Lists all currently opened databases.
setinfo	Sets database information, such as its name, version number, and modification number.

Debugging Utility Commands

dm	Displays memory.
gdb	Enables or disables Gdb debugging.
mdebug	Sets the Memory Manager debug mode.
sb	Sets the value of a byte.

Gremlin Commands

gremlin	Activates the specified gremlin until a specified event occurs.
gremlinoff	Deactivates the current gremlin.

Heap Utility Commands

hc	Compacts a memory heap.
hchk	Checks a heap.
hd	Displays a dump of a memory heap.
hf	Allocates all free space in a memory heap, minus a specified number of bytes.
hi	Initializes a memory heap.
hl	Lists all of the memory heaps on the specified memory card.
hs	Scrambles a heap.
ht	Performs a heap total.
htorture	Torture-tests a heap.

Host Control Commands

help	Provides help on the console commands.
log	Starts or stops logging to a file.
saveimages	Saves an image of a memory card to file.

Miscellaneous Utility Commands

simsync	Simulates a synchronization operation on a database.
sysalarmdump	Displays the alarm table.

Record Utility Commands

addrecord	Adds a record to a database.
attachrecord	Attaches a record to a database.
changerecord	Replaces a record in a database.
delrecord	Deletes a record from a database.
detachrecord	Detaches a record from a database.
findrecord	Finds a record by its unique ID.
listrecords	Lists all of the records in a database.
moverecord	Changes the index of a record.
setrecordinfo	Sets record information, such as its ID and attributes.

Resource Utility Commands

addresource	Adds a resource to a database.
attachresource	Attaches a resource to a database.
changeresource	Replaces a resource in a database.
delresource	Deletes a resource from a database.
detachresource	Detaches a resource from a database.
getresource	Retrieves a resource from a database.
listresources	Lists all resources in a database.
setresourceinfo	Sets resource information, such as its ID and resource type.

System Commands

battery	Battery utility command for starting or stopping radio charging, and for setting the loaded status.
coldboot	Boots the handheld device.
doze	Puts the CPU to sleep while keeping the peripherals and clock running on the handheld device.
exit	Exits the console.
feature	Displays, retrieves, registers, or unregisters features.
kinfo	Displays kernel information.
launch	Launches an application.
performance	Sets performance levels, such as the system clock frequency and CPU duty cycle.
poweron	Powers on the handheld device.
reset	Resets the memory system and formats both cards.
sleep	Shuts down all peripherals, the CPU, and the system clock.
switch	Switches the current user interface application.

5

Using Palm Reporter

This chapter describes Palm Reporter, which you can use to do trace analysis of your Palm OS® applications. The following topics are covered in this chapter:

- "About Palm Reporter" - An introduction to Palm Reporter concepts
- "Downloading Palm Reporter" on page 162 - How to download and install the Palm Reporter package
- "Adding Trace Calls to Your Application" on page 163 - How to add Host Control trace calls to your application
- "Displaying Trace Information in Palm Reporter" on page 165 - How to open a Palm Reporter session to view the trace information
- "Troubleshooting Palm Reporter" on page 169 - How to make sure Palm Reporter is running correctly

About Palm Reporter

Palm Reporter is a trace utility that can be used with Palm OS Emulator. As an application runs on Palm OS Emulator, it can send information in real time to Reporter. This information can help pinpoint problems that might be hard to identify when executing code step-by-step or when specifying breakpoints. To view the realtime traces, simply run Reporter at the same time as you run your application on Palm OS Emulator.

Palm Reporter Features

Palm Reporter has a number of features that make it useful:

- High throughput of trace output, allowing for realtime traces
- Trace output filtering, searching, saving, printing, and copying

- Display of Trace output through a TCP/IP connection

Downloading Palm Reporter

The most recent released version of Palm Reporter is posted on the internet in the Palm™ developer zone:

http://www.palmos.com/developers

Follow the links from the developer zone main page to the Palm OS Emulator page to retrieve the released version of Palm Reporter.

Palm Reporter Package Files

The Palm Reporter package includes the following files:

Table 5.1 Files Included in the Palm Reporter Package

File	Description
Windows: `Reporter.exe` Macintosh: `Reporter`	Main Palm Reporter program file
Windows: `PalmTrace.dll` Macintosh: `PalmTraceLib`	Palm OS Emulator add-on that relays traces to Palm Reporter
`TraceTest.prc`	Sample application containing HostTrace API calls
`Documentation` (folder)	Palm Reporter documentation, including: • `Reporter guide.html` • `Reporter protocol.html`

Installing Palm Reporter

Palm Reporter requires Palm OS Emulator. Place the `PalmTrace` library (`PalmTrace.dll` or `PalmTraceLib`) in the same folder as the Palm OS Emulator executable. Emulator will not be able to send trace information to Reporter if it cannot find and load the `PalmTrace` library.

The Palm Reporter executable can be located in any folder on your system; it does not need to be in the same folder as Palm OS Emulator.

Adding Trace Calls to Your Application

Traces are generated by system calls that are recognized by Palm OS Emulator but ignored by actual handheld devices. These system calls are listed in `hostcontrol.h`, which is part of both the Palm OS SDK and the Palm OS Emulator package. For more information about the Host Control API, see the book *Using Palm OS Emulator*.

The Host Control system calls pertinent to tracing are listed in the following table:

System Call Format	Function Description
`void HostTraceInit(void)`	Initiate a connection to Reporter
`void HostTraceOutputT(UInt16 mod, const char* fmt, …)`	Output a string to Reporter (printf format)
`void HostTraceOutputTL(UInt16 mod, const char* fmt, …)`	Output a string to Reporter (printf format) with an additional line break
`void HostTraceOutputB(UInt16 mod, const char* buff, UInt32 len)`	Send binary data to Reporter
`void HostTraceOutputVT(UInt16 mod, const char* fmt, va_list vargs)`	Output a string to Reporter (vprintf format)
`void HostTraceOutputVTL(UInt16 mod, const char* fmt, va_list vargs)`	Output a string to Reporter (vprintf format) with an additional line break
`void HostTraceClose(void)`	Close the connection to Reporter

All `HostTraceOutput` functions take an error class identifier as their first parameter. This parameter allows filtering of traces according to their origin. Recognized error classes are listed in

`SystemMgr.h`. For example, applications should specify the error class `appErrorClass`.

Specifying Trace Strings

Trace strings use the following format:

```
% <flags> <width> <type>
```

`<flags>`

-	Left-justify display (Default is right justify)
+	Always display the sign character (Default is to display the sign character for negative values only)
space	Display a space (when a value is positive) rather than displaying a "+" sign
#	Alternate form specifier

`<width>` Must be a positive number

`<type>`

%	Display a "%" character
s	Display a null-terminated string
c	Display a character
ld)	Display an Int32 value
lu	Display a UInt32 value
lX or lx	Display an Int32 or UInt32 value in hexadecimal
hd	Display an Int16 value
hu	Display a UInt16 value
hX or hx	Display an Int16 or UInt16 value in hexadecimal

NOTE: The following types are not supported for `<type>`: o, e, E, f, F, g, G, p, l, n, d, i, u, X, or x.

Trace Functions in a Code Sample

```
void function(void)
{
unsigned char theBuffer[256];
unsigned long theUInt32 = 0xFEDC1234;
unsigned short theUInt16 = 0xFE12;
int i;

HostTraceInit();

HostTraceOutputTL(appErrorClass, "This is an Int32:");
HostTraceOutputTL(appErrorClass, " unsigned (lu) [4275835444]=[%lu]",
theUInt32);
HostTraceOutputTL(appErrorClass, " signed (ld) [-19131852]=[%ld]", theUInt32);
HostTraceOutputTL(appErrorClass, " hexa (lx) [fedc1234]=[%lx]", theUInt32);

HostTraceOutputTL(appErrorClass, "This is an Int16:");
HostTraceOutputTL(appErrorClass, " unsigned (hu) [65042]=[%hu]", theUInt16);
HostTraceOutputTL(appErrorClass, " signed (hd) [-494]=[%hd]", theUInt16);
HostTraceOutputTL(appErrorClass, " hexa (hX) [FE12]=[%hX]", theUInt16);

HostTraceOutputTL(appErrorClass, "This is a string (s) [Hello world]=[%s]",
"Hello world");
HostTraceOutputTL(appErrorClass, "This is a char (c) [A]=[%c]", 'A');

HostTraceOutputTL(appErrorClass, "This is a buffer:");

for (i = 0 ; i < 256 ; i++) theBuffer[i] = (unsigned char) i;

HostTraceOutputB(appErrorClass, theBuffer, 256);

HostTraceClose();
}
```

Displaying Trace Information in Palm Reporter

To view trace information in Palm Reporter, you need to do the following:

- Add trace calls to your application and build your application
- Start a Palm Reporter session
- Start a Palm OS Emulator session
  - Set the Emulator "Tracing Options" to display output to "Palm Reporter"
  - Install your trace-enabled application in the Emulator session
  - Run your trace-enabled application in the Emulator session

Starting a Palm Reporter Session

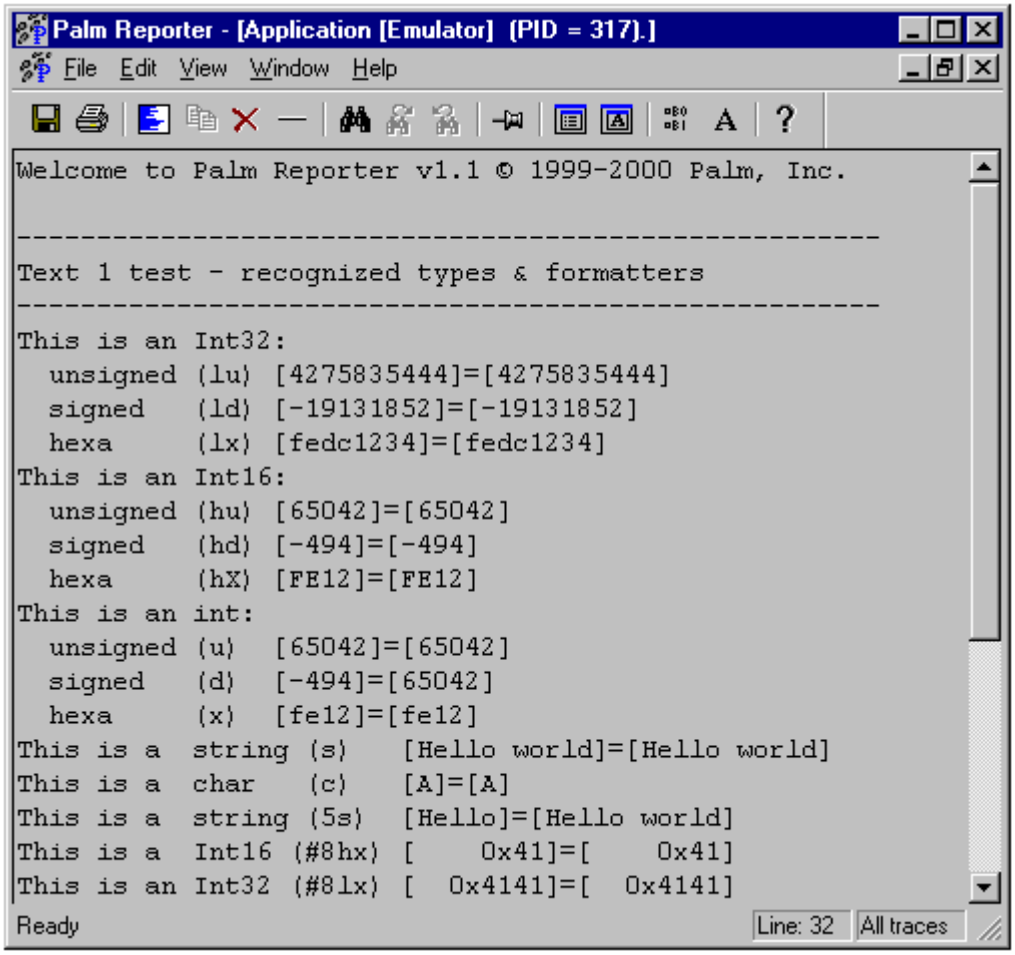
To start a Palm Reporter session, run the `Reporter.exe` file. After starting Palm Reporter, you should see an empty window. This window will serve as a container for other windows which display the trace information. A new trace window is created for each `HostTraceInit` to `HostTraceClose` sequence in your trace-enabled application.

Each `HostTraceOutput` call will send information into the current trace window. The `HostTraceOutput` call will fail if there is no active trace window, which can happen if Reporter is not running when the `HostTraceInit` function is called.

Also, a reset in Emulator will close any pending connection. That is, Emulator will call the `HostTraceClose` function for your application if you used `HostTraceInit` to open a trace connection.

Figure 5.1 shows a Palm Reporter session window.

Figure 5.1 Palm Reporter Session Window



Filtering Information in a Palm Reporter Session

You can control the type of trace information Palm Reporter displays. You control this information by setting *filters*. Filters can be set either globally, by using the **Global filters...** menu, or for the current window, by using the **Active view filters...** menu. By enabling or disabling the filters, you can choose to view traces sent by corresponding modules in your application. Global filter settings are saved when you exit the Palm Reporter session.

Using the Palm Reporter Toolbar

Palm Reporter provides a toolbar with the following functions:

Toolbar Icon	Function
	Save the contents of the Reporter window to a file
	Print the contents of the Reporter window
	Select all of the text in the Reporter window
	Copy the selected text into the system clipboard
	Clear the contents of the Reporter window
	Draw a horizontal line across the Reporter window
	Search the Reporter window for specified text
	Search the Reporter window for the next occurrence of specified text
	Search the Reporter window for the previous occurrence of specified text
	Set “on top” mode to keep the Reporter window always visible on the screen
	Set filters for the current window only
	Set font for the current window only

Toolbar Icon	Function
	Set filters for all new windows
	Set font for all new windows

Troubleshooting Palm Reporter

Table 5.2 How to Solve Possible Palm Reporter Problems

Symptom	Solution
You are unable to set the Emulator "Tracing Options" to display output to "Palm Reporter".	Make sure that the `PalmTrace` library is in the same folder as the Palm OS Emulator executable.
The `PalmTrace` library (`PalmTrace.dll` or `PalmTraceLib` file) doesn't appear in the folder where you decompressed the Reporter's archive.	Check to see if your system is configured to "Hide system files."

Table 5.2 How to Solve Possible Palm Reporter Problems

Symptom	Solution
Nothing appears in the Palm Reporter session window.	Make sure that: • The `PalmTrace` library (`PalmTrace.dll` or `PalmTraceLib` file) is in the same folder as the Palm OS Emulator executable. • Your application code is calling `HostTraceInit`. • You are using Palm OS Emulator version 3.0a4 or later. • You have set the Emulator "Tracing Options" to display output to "Palm Reporter". • Your filters are set correctly, and traces are emitted with the right modules.
You have checked everything in this table, and Reporter still isn't displaying trace information.	Send a note describing your problem to `reporter@palm.com`.

Table 5.3 Palm Reporter Error Message

Error Message	Problem	Possible Solution
An error occurred while trying to listen for traces.	Default reception port is already in use.	Check that no other instance of the Reporter is running.
An error occurred while initializing ObjectSet.	Framework initialization failed.	Send a note describing your problem to `reporter@palm.com`.
An error occurred while ObjectSet was initializing TCP/IP.	TCP/IP related failure.	Check that TCP/IP networking is correctly set up.

Table 5.3 Palm Reporter Error Message *(continued)*

Error Message	Problem	Possible Solution
Cannot load filters description.	The Reporter executable file was altered.	Send a note describing your problem to `reporter@palm.com`.
Unable to start a reader thread.	Reporter could not create receiver thread.	Free up system resources.
Unable to start a format thread.	Reporter could not create displayer thread.	Free up system resources.

6

Using the Overlay Tools

This chapter describes how the PRC-to-Overlay tools can be used to produce a localized version of an application. The following topics are covered in this chapter:

- "Using Overlays to Localize Resources"- An overview of using overlay databases to localize application resources.
- "About the Overlay Tools" on page 175 - An introduction to the PRC-to-Overlay and Patch Overlay tools.
- "Using the PRC-to-Overlay Function" on page 175 describes how to create overlay resource databases for localized data.
- "Using the Patch Overlay Function" on page 178 describes how to use multiple overlay resource databases with a single bases application database.
- "PRC2OVL Options Summary" on page 179 lists the command line options used with `PRC2OVL`.
- "Using PRC2OVL on the Macintosh" on page 181 contains special instructions for using `PRC2OVL` on a Macintosh system.

Using Overlays to Localize Resources

Palm OS® 3.5 added support for localizing applications through overlay databases. Each overlay database is a separate resource database that provides an appropriately localized set of resources for a single base database (a `PRC` file) and a single target locale (language and country).

Support for overlay databases is provided by Overlay Manager. To use Overlay Manager, create a base application that has your base resources (usually English) for your user interface and a separate overlay database that has the substitutions you want to make for

each locale (French, German, Japanese, etc.). When an application runs on a localized version of Palm OS, Overlay Manager automatically substitutes localized resources from the appropriate overlay database at runtime. Alternatively, you can use Data Manager routine `DMOpenDBWithLocale()` to open a base database with an arbitrary overlay.

For more information about Overlay Manager and localizing your applications, see *Palm OS Programmer's Companion*.

Overlay Database Names

Each overlay database name contains a *locale suffix*. A locale consists of a language indicator and a country code:

- The first two letters indicate the language and must be lower case.
- The second two letters indicate the country and must be upper case.

For example, the database name `Address Book_enUS.PRC` indicates that this is an overlay for the language "English" and the country "United States."

Overlay Specification Resources

Overlay specification resources establish a link between the base and the overlay databases. They bind resources together and are important when you have multiple version of the same database (for example, version 1 and version 2 of an application). Overlay specifications are required for overlay databases, but optional for the base database.

Overlay specification resources contain the following information:

- Type information (`'ovly'` for overlay databases)
- `ID = 1000`
- Target locale (language and country)
- Information about the base database (type, creator, checksum, etc.)
- Information about each overlaid resource. This content specifies exactly which resources are overlaid. Normally, this

content consists of replacements for resources in the base, but it can also specify additional resources that are not in the base.

About the Overlay Tools

The overlay tools allow you to produce an overlay database that can be superimposed on top of another so that any requests for the underlying base database first go through the overlay database. This allows localization to be performed by placing the localized (for example, German) data in an overlay for a particular locale (for example, Germany).

You can edit and distribute the overlay separately from the underlying database. Because the overlay only needs to contain localized data, it does not need to include your application code or other large resources.

Using the PRC-to-Overlay Function

The PRC-to-Overlay function takes a normal resource database (usually an application) as input and produces an overlay. You can also give the tool an overlay as input to create a new overlay for a different locale.

How the PRC-to-Overlay Function Works

The PRC-to-Overlay function takes a single file as input, passes the file through a set of filters to decide which particular resources (components of the database) are localizable and should be put in the overlay. Then, given a particular locale, the tool generates an overlay file.

Choosing a Locale

A locale consists of a language indicator and a country code:

- The first two letters indicate the language and must be lower case.

- The second two letters indicate the country and must be upper case.

To list the available language and country codes, use the following command:

```
prc2ovl -showlocales
```

For example, the following command creates an English language overlay for the country United States (using the default filter set):

```
PRC2OVL NewApp.prc -locale enUS -o
NewApp_enUS.prc
```

where:

`NewApp.prc` Indicates the input file name "`NewApp.prc`"

`-locale enUS` Indicates the language code is "`en`" for English and the country code is "`US`" for United States

`-o NewApp_enUS.prc`
Specifies the output file name "`NewApp_enUS.prc`"

Modifying the Filter Set

A filter set indicates which particular resources (components of the database) are localizable and which resources should be put in the overlay `PRC`.

To modify the filter set, use the `-a`, `-n`, `-i`, and `-e` switches:

`-a` indicates that all resources are to be localized.

`-n` indicates that no resources are to be localized.

`-i` includes a particular set of resources (in the list of localized resources).

`-e` excludes a particular set.

Each switch operates in the order in which it appears on the command line. The last switch that matches is the one that is operated on. For example, the filter set:

```
-n -i tFRM 1000
```

produces an overlay that only contains the single `'tFRM 1000'` resource (if it is present in the input), but the filter set:

```
-a -e tFRM 1000
```

localizes everything but the `'tFRM 1000'` resource.

Default Filters

Recreate the default filters with the following set of parameters:

```
-a -e CODE -e DATA -e code -e data
-e boot -e extn -e pref
```

Restricting Resource Matches

You can restrict matches by ID number. For example, if you only want to localize resource type `'BAZZ'` with ID 567, specify the filter set:

```
-i BAZZ 567
```

You can also supply ranges in your filter set, as shown in the following example:

```
-i BAZZ 567-599
```

Note: To see which resources are selected in the output, use the `-v` (for verbose) switch.

PRC2OVL Example

This example shows the files that are included as part of an application that needs to be localized.

The `NewApp.prc` file contains the application named NewApp which is written in English. The `PRC` file contains the following resources:

- `Resource 0: 'CODE' 0`, application code
- `Resource 1: 'CODE' 1`, more application code
- `Resource 2: 'tFRM' 1000`, application form
- `Resource 3: 'tSTR' 1000`, UI strings

Using the following command:

```
PRC2OVL NewApp.prc -locale deDE -o
NewApp_deDE.prc
```

Creates a German overlay, `NewApp_deDE.prc`, which is a file containing the following resources:

- `Resource 0: 'tFRM' 1000`, application form
- `Resource 1: 'tSTR' 1000`, UI strings

Using the Patch Overlay Function

The Patch Overlay function takes two input files, a base `PRC` and an overlay `PRC`, and outputs a new overlay `PRC` that has been modified so it will work with the given base `PRC`. This is accomplished by copying the appropriate data over the overlay resource in the overlay file, synthesizing necessary data if the base `PRC` was stripped.

You specify the Patch Overlay function with the `-p switch`. For example,

```
PRC2OVL OrigGermanOvl.prc -c
-p EnglishBase.prc -o FixedGermanOvl.prc
```

where:

`OrigGermanOvl.prc`
: Indicates the input overlay `PRC` filename.

`-c`
: Indicates whether to generate a new checksum for the output overlay `PRC`.

  If you omit the "`-c`" parameter, then `PRC2OVL` will copy appropriate data over the overlay resource in the overlay file, synthesizing necessary data if the base `PRC` was stripped, and will generate a new checksum for the output overlay `PRC`.

If you include the "-c" parameter, then PRC2OVL will simply generates a new checksum for the output overlay PRC, without copying data over the overlay resource in the overlay PRC.

`-p EnglishBase.prc`
Indicates this is a Patch Overlay function and `EnglishBase.prc` is the input base PRC filename.

`-o FixedGermanOvl.prc`
Indicates the output overlay PRC filename.

Example

This example shows how you could build two language versions as separate projects, and generate two language overlays that would work for a single base:

1. Build your English language project: `EnglishApp.prc`.
2. Create a second project, where you duplicate the code from the first project, but change the resources for your desired localization. For example: `GermanApp.prc`.
3. Use PRC-to-Overlay to generate an English overlay: `EnglishOvl.prc`.
4. Use PRC-to-Overlay to generate a German overlay: `GermanOvl.prc`.
5. Use the Patch Overlay function to incorporate the checksums and overlay resource descriptions from the English application into the `GermanOvl.prc`, calling it `FixedGermanOvl.prc`.

As a result, you would have an `EnglishBase.prc` that would work with two overlay PRCs: `EnglishOvl.prc` and `FixedGermanOvl.prc`.

PRC2OVL Options Summary

The following tables list the PRC2OVL command line options. These options can be specified in any order.

Table 6.1 PRC2OVL Options for the PRC-to-Overlay Function

Option	Description
`-h`	Display help information.
`-o` *filename*	Specify the name of output file.
`-showlocales`	List the available language and country codes.
`-locale` *llCC*	Specify a locale code, where `ll` indicates the language and `CC` indicates the country code.
`-a`	Specify a filter set that localizes all resources.
`-n`	Specify a filter set that localizes no resources.
`-i` *resourceID(s)*	Specify a filter set that includes a particular set of resources, where *resourceID(s)* can be a single resource ID number (for example, 567) or a range of resource ID numbers (for example, 567-599).
`-e` *resourceID(s)*	Specify a filter set that excludes a particular set of resources, where *resourceID(s)* can be a single resource ID number (for example, 567) or a range of resource ID numbers (for example, 567-599).
`-v` or `-V`	Print status information to the screen.

Table 6.2 PRC2OVL Options for the Patch Overlay Function

Option	Description
`-h`	Display help information.
`-c`	Generate a new checksum for the output overlay `PRC`, without copying data over the overlay resource in the overlay `PRC`.
`-p` *filename*	Specify the name of the input base `PRC` file.
`-o` *filename*	Specify the name of output overlay `PRC` file.

Getting Help

You can get help when you:

- Run `PRC2OVL` (or `MPWPRC2OVL`) without arguments.
- Enter invalid arguments.
- Use `-h` on the command line.

Help lists the default resource selection filters.

Using PRC2OVL on the Macintosh

This section describes how to use `PRC2OVL` on a Macintosh graphical user interface (GUI).

Opening a PRC file

You can use the Mac GUI to create an overlay for a `PRC` file; typically the `PRC` file contains an application or a preference panel. Open the `PRC` file, then pick a target locale (which is the same as the `-locale` switch). The application displays the entire list of resources in the file, using the same default selection criteria, if necessary, to provide a suggested set of resources to localize. You can edit these by clicking on the checkbox by each item in the list. Then you can build an output file by clicking on the Build button.

Selecting Resources

The Mac GUI tool lets you select the resources you want to localize from a list rather than specifying resources with filters on the command line. By default, the tool assumes that all resources are overlaid except those of types `'CODE'`, `'DATA'`, `'code'`, `'data'`, `'boot'`, `'extn'`, and `'pref'`. (You can select other resources via the filter options you use in the command-line tool.)

A

Resource Tools

There are two tools provided with the Metrowerks CodeWarrior environment that you can use to work with resources:

- Use the Rez tool to compile a textual description of the resources for your application into a resource file.
- Use the DeRez tool to decompile a resource file into a text file.

Both of these tools are standard Apple Computer tools for working with Macintosh OS application resources. Documentation for both the Rez and DeRez programs is found in the Apple book *Building and Managing Programs in MPW, 2nd Edition*. This book is available online at the following URL:

http://developer.apple.com/tools/mpw-tools/books.html

B

Simple Data Types

Table B.1 describes the simple data types, which have been renamed in the newest release of the Palm OS® software.

Table B.1 Simple Data Types

Old data type name	New data type name	Description
`Byte`	`UInt8`	unsigned 8-bit value
`UChar`	`UInt8`	unsigned 8-bit value
`SByte`	`Int8`	signed 8-bit value
`Int`	`Int16`	signed 16-bit value
`SWord`	`Int16`	signed 16-bit value
`Short`	`Int16`	signed 16-bit value
`UShort`	`UInt16`	unsigned 16-bit value
`UInt`	`UInt16`	unsigned 16-bit value
`Word`	`UInt16`	unsigned 16-bit value
`Long`	`Int32`	signed 32-bit value
`SDWord`	`Int32`	signed 32-bit value
`ULong`	`UInt32`	unsigned 32-bit value
`DWord`	`UInt32`	unsigned 32-bit value
`Handle`	`MemHandle`	a handle to a memory chunk
`VoidHand`	`MemHandle`	a handle to a memory chunk
`Ptr`	`MemPtr`	a pointer to memory
`VoidPtr`	`MemPtr`	a pointer to memory

Index

Symbols

A

B

C

D

E

F

G

H

R

S

T

U

V

W