Handspring Development Tools Guide

Release 1.2

Document number: 80-0092-00

Handspring, Inc.

189 Bernardo Ave.

Mountain View, CA 94043-5203

TEL: (650) 230-5000

FAX: (650) 230-2100

www.handspring.com

Table of Contents

1 Introduction

1.1 Intended Scope of this Document

This guide is intended primarily for developers who want to create applications for Handspring products including specialized Springboard™ modules. Currently, Software Development Kits (SDKs) designed to assist developers are developed using the Handspring™ Palm OS® GNU Tools. These tools are discussed in this document. In addition, utilities such as Palm-MakeROM, which are part of the Handspring GNU Tools, are documented here for developers who want to port Palm OS based software to a Springboard module.

For Palm OS software development, the following section provides a list of useful resources; however, the remainder for this guide will focus on software tools required to develop a Springboard module.

Software for the Handspring systems can be developed in the same way as any Palm OS-based application; however, when you want to create a Springboard module for your applications, you must then use Handspring's development tools.

1.2 Types of Software Development for Handspring Handhelds

The key factor that makes the Springboard™ Expansion Slot a compelling platform is its *plug-and-play functionality*. The Springboard Expansion Slot allows different modules to be inserted and removed from a handheld computer at *any time*. To support this functionality, Handspring has created extensions to the standard Palm OS® in order to enable new, specialized hardware and software to:

- Detect the insertion of a module.
- Load applications and appropriate drivers stored on the module.
- Cleanly remove software when the module is removed.

This functionality correctly implements plug-and-play for the handheld computer.

Application software that resides in a module's memory is executed in place, just like applications in the device's internal ROM or RAM. With this design, Palm OS jumps directly to program code, rather than "loading" an

application into internal memory and "jumping" to the appropriate code on the module memory. This functionality is sometimes referred to as "execute in place." This architecture is well suited to handheld devices in which memory and processing power are scarce resources.

The Springboard Expansion Slot builds on this architecture by directly mapping memory and I/O ports on the module to the main processor's address space. Access to hardware in the handheld and on the module is conducted in exactly the same way. Software is executed in place on the module. To support full plug-and-play functionality, the system allows for removal of a module while it is running an application. The user is automatically switched out of the module application and back to the Application Launcher, as necessary.

This guide describes the Springboard Expansion Slot, and provides you with the information necessary to:

- Implement "application-only" products on a Springboard-compatible memory module.
- Design Springboard-compatible modules with specialized hardware and the applications to support them.

1.3 Software Development for Handspring Handhelds

There are three general categories of software development for Handspring handhelds. Each category has different requirements. Each of these categories is described in greater detail in the sections that follow. The three categories are:

Generic Applications: These applications execute from the Handspring handheld's internal memory.

> *Examples:* Any of the utilities, games, and other applications that can be downloaded for use on Palm OS-based devices.

Generic Applications on a Springboard Module: These applications execute from a Springboard memory module. Inserting a module provides instant access to the application, eliminating the need to download and install software.

> *Example:* A Palm OS reference guide that is distributed on a Springboard module to accommodate retail distribution. In particular, a Springboard memory module is a better platform for larger applications that would be cumbersome to install and consume internal memory.

Special Purpose Applications: These applications access specialized hardware on the Springboard module. Additionally, a "special purpose" application may also install interrupt handlers and other system modifications in support of the module hardware. All the software necessary to operate the module is resident on the module itself, eliminating the need to download and install software and drivers.

> *Example*: Software and drivers to operate a Springboard module such as VisorPhone (Handspring's GSM phone module).

1.4 Generic Applications

There are various development environments for Palm OS-based systems. The two primary tools are the Palm OS GNU Tools and Metrowerks' CodeWarrior. Following is an overview of these tools. For more detailed information, or to download these tools, go to the Handspring website at:

http://www.handspring.com/developers/sw_dev.jhtml

1.4.1 Handspring Palm OS GNU Tools

The GNU Tools are based on the Unix environment and are command line driven. If you're accustomed to Unix development, this will be very familiar. To develop for Handspring handhelds you must use Handspring's GNU Tools that use the Cygwin GNU utilities for Windows. Handspring has also developed a Microsoft Visual C++ project file that calls the GNU Tools. With this approach, you can have the ease of using an IDE with the flexibility of open source development tools. A complete description of the tools is included with the download. These tools are currently available for download from Handspring for use in the Windows environment. The Handspring Extension header files (available from Handspring's website) must be installed onto this development tool for applications using the Handspring Extension APIs.

1.4.2 Metrowerks CodeWarrior

Metrowerks CodeWarrior is an integrated development environment with a graphical interface that allows for easier generation of forms. It also provides a utility for automatically managing files and resources. The "lite" version -- which doesn't allow for software distribution -- is free to download. The full version can be purchased from Metrowerks. These tools are currently available from Palm for use in both the Windows and Macintosh environments. The Handspring Extension header files (available from Handspring's website) must be installed onto this development tool for applications using the Handspring Extension APIs.

1.4.3 Other Development Environments

The Palm developers' web site (http://www.palmos.com/dev/tech/tools/) contains information on other development environments available for the Palm OS.

Developing software for Handspring handhelds is the same as developing for other Palm OS-based systems. For example, Visor™ is based on Palm OS 3.1, so documentation covering standard Palm OS development is applicable. The Palm developers' web site (http://www.palmos.com/dev/support/docs/) contains a variety of references covering standard Palm OS development.

1.5 Generic Applications on a Springboard Module

If you are an application developer who simply wants to transfer your application to a non-volatile Springboard memory module, you simply need the Palm-MakeROM tool as described in this guide to build a ROM image. Third party suppliers can use your ROM image to program memory modules in quantity.

For development purposes, you can also use Handspring's 8MB Flash Module. This module, available on our web site, is a run-time read-only memory-based module that can be re-programmed using the Palm Debugger. Handspring includes the FileMover application with this module that enables users to transfer any applications between internal and module memory. Developers can use the FileMover application to move software onto the module for testing purposes.

Handspring has also developed CardUpdaterMakerSDK that provides an easy way for developers to generate a utility that customers can use to update a Flash module. Note that this process will erase the entire flash card and it will not be usable as an 8MB Flash Module. To restore the original functionality of this module you will need to download the appropriate file from our Customer Care website.

http://www.handspring.com/support/index.jhtml

Since the Springboard memory modules can be removed at any time during execution, there are some considerations to take into account when designing your application. Specifically, software that uses shared libraries or installs system modifications (e.g., interrupt handlers) should be configured properly to work with the plug-and-play features of the Springboard Expansion Slot. Refer to the section entitled *Application Development To Support Plug-and-Play* in the Springboard Development Guide for a more detailed description.

You might want to consider designating a Welcome application on your module. This application would be launched automatically when your Springboard module is inserted in a handheld. Refer to the section entitled *Module Welcome application* in the Springboard Development Guide for a more detailed description.

Finally, applications that execute from a Springboard memory module (i.e., masked ROM, Flash, and OTP) are based on read-only memory and should be designed appropriately.

1.6 Special Purpose Applications

If you are building a Springboard module with specialized hardware, you must use the Handspring Extension API. This API and all other necessary information to build Springboard-compatible modules are explained in the next sections.

The Developer section of Handspring's website contains source code examples for Springboard module applications. They have been developed using the Handspring Palm OS GNU Tools. These examples show how to develop more sophisticated applications that install interrupt handlers or OS patches when the associated module is plugged into the Springboard Expansion Slot. These examples have been fully tested at Handspring and can be used as a baseline for application development.

2 Handspring Coding Standards

2.1 About the Handspring Coding Standards

The Handspring coding standards are essentially the Palm OS naming conventions, combined with GNU formatting conventions. Developers will find all Handspring code in this format. These conventions are recommendations for easier readability and are not requirements.

2.2 General Program Design

In general, the following guidelines should be followed to make your code as bug-free, maintainable, and portable as possible:

- **Minimize the use of global variables**. This falls under the general guideline of using good data abstraction. In general, try to minimize the exposure of any variables to the smallest subset of functions that need to directly access the variable. When performance is acceptable, require the use of accessor functions to read and/or modify variables.
- **Minimize the number of exported functions and equates.** Try to make as many functions and equates private (static) to a particular source code module as possible.
- **Choose long, descriptive function, variable, and equate names.** This helps document the code and can save you from having to write as many comments.
- **Make the code look "pretty**". Judicious use of white space and comment blocks can go a long way to making your code more readable and maintainable -- especially by others.
- **Separate platform-independent code into separate source files from platform-dependent code.** Always try to structure your sources so that only a subset of the source files need to be rewritten when porting to a different platform. For example, all the code that makes Win32 API calls for a GUI Windows application should be broken out into separate source files from the core platform-independent code.
- **Avoid the use of arbitrary limits on the length of any array including filenames, symbols, and user input**. Allocate these types of structures dynamically.
- **Always check result codes on function calls.** When compiled for debug mode, display the filename, line number, and associated error text for any error encountered.

2.3 Organization of Source Files

Each project should be in it's own directory with the following structure and directory names:

```
/MyProject
    /Build                # makefiles, IDE project files, build scripts
    /Docs                 # Documentation, including:
       ChangeLog.txt      # text file with change notes
       Requests.txt       # text file with feature and bug fix requests
       *.html             # HTML documents
       *.doc              # Word documents
    /Incs                 # Include files that are shared among the different
                          # source files in the project. Private includes should
                          # be put into the /Src directory.
    /Obj                  # all object code, temp files, and the final executable
                          # go here. Basically, any files that can be deleted
                          # and re-built should go here.
    /Src                  # platform-independent source code
    /SrcWin               # Windows-specific source code
    /SrcMac               # Macintosh-specific source code
```

2.4 Formatting Conventions

All source files should be edited using a tab stop of 4, and lines should be no wider than a typical portrait-size window (approximately 80 columns).

All function implementations should be formatted as shown below with the return type, function name, and open and close braces all at column 0. If the parameters must be continued on the next line, align them with the first parameter of the line above. The body of the function starts at column 2.

```
void*
GdbExecuteCmd (void* gP, char** resultsPP, unsigned int echoReq,
               unsigned int echoRsp, long timeout)
{
  // function body goes here
  return MyFunction (gP);
}
```

When declaring pointer types, place the asterisk immediately after the base type, not next to the pointer name. Using this convention, multiple pointer variables must be declared on separate lines, such as:

```
  char* srcP;
  char* dstP;
```

When making function calls, put spaces before the open parenthesis, and after the commas:

```
  MyFunction (param1, param2, param3);
```

Inside the body of the function, if-else statements should appear with braces indented two columns and the body of the "if" indented two more columns. If there are no braces, then the single line body is simply indented two columns:

```
  if (gP == 0)
    foo = bar;
  else
    {
      foo = bar + 2;
      return foo;
    }

  if (myVar == 0)
    foo = bar;
  else if (myVar == 1)
    {
      foo = bar + 1;
      return foo;
    }
```

Try to avoid assignments within conditional tests. Instead, include an extra assignment line as follows:

```
foo = MyFunction();
if (foo == 0)
  Exit();
```

While and do-while loops should be formatted as follows:

```
while (foo)
  {
    bar = MyFunction (bar);
    foo--;
  }

do
  {
    bar = MyFunction (bar);
    foo--;
  }
while (foo > 3);
```

When conditional tests or evaluations must repeat on the next line, split them before an operator -- not after -- as shown below. Also, try to avoid having two operators of different precedence at the same level of indentation:

```
if ((thisIsLong && thisIsLong2 && thisIsLong3 && thisIsLong4
     && thisIsLong5 || (anotherLong && anotherLong2))
    || anotherLong3)
  {
    thisVariable = variable1 + variable2 + variable3
                   + variable 4;
  }
```

Switch statements should be formatting as follows:

```
switch (foo)
  {
  case 1:
    DoSomething ();
    break;
  case 2:
    DoSomethingElse ();
    break;
  }
```

2.5 Commenting and Style

The top of every source and header file should contain a comment section formatted such as:

```
/*****************************************************************************
 *
 * Project:
 *    PalmDebugger
 *
 * Copyright info:
 *    <developer Copyright notice goes here>
 *
 * FileName:
 *    GdbWin.cpp
 *
 * Description:
 *    This file contains the code that launches and communicates with the
 *    background process running gdb under Windows.
 *
 * ToDo:
 *
 * History:
 *    21-Nov-1998 - Created by AB
 *
 *****************************************************************************/
```

The implementation of each function should be preceded by a comment section formatted as shown below. It should contain the function name, a summary of what the function does, the list of parameters, with each identified as either an input (IN), output (OUT), or input and output (INOUT), a description of the possible return values, a list of the most likely callers of this function, miscellaneous notes, and a history section.

```
/************************************************************************
 *  Function:    GdbExecuteCmd
 *
 *  Summary:
 *    Sends a command string to the GDB process and waits and
 *    optionally captures all output from the command.  This
 *    routine will automatically append a newline to the end of
 *    the command string if not present already. It will also
 *    strip off the command prompt 'prvPromptStr' if present
 *    at the beginning of the command string.
 *
 *  Parameters:
 *    gP          IN    Pointer to private GDB Globals
 *    *resultsPP  OUT   If resultsPP is not null, then on exit *resultsPP
 *                      will contain a malloc'ed string with the results
 *                      of the GDB cmd.
 *    echoReq     IN    if true, echo command to the GDB UI window.
 *    timeout     IN    timeout (in ticks) to wait for response
 *                      string from GDB. If 0 timeout, then
 *                      this routine returns without waiting for
 *                      a response.
 *
 *  Returns:
 *    0 if no error
 *
 *  Called By:
 *    AppExecuteCmdLine
 *
 *  Notes:
 *    This routine could be optimized better by blah, blah, blah...
 *
 *  History:
 *    9-Dec-1998  AB  Created
 ************************************************************************/
```

In general, use white space and comment lines intelligently to help illustrate the structure of a function. A cleanly formatted source file helps tremendously in debugging and maintenance -- especially when being used by someone other than the original author.

For example:

```
Int
PrvSendStr (PrvGlobalsType* gP, CharPtr cmdP)
{
  DWord           bytesWritten;
  DWord           err;


  // ------------------------------------------------------
  // Error check. These conditions have to be checked beforehand by
  //  the caller (PrvExecuteCmd).
  // ------------------------------------------------------
  assert (!gP->stdio.needInitCmds);
  assert (!gP->stdio.needRestart);


  // ------------------------------------------------------
  // Send the command line to GDB's stdin now.
  // ------------------------------------------------------
```

```
  if (!WriteFile (gP->stdio.stdInWriteH, cmdP, strlen (cmdP),
            &bytesWritten, NULL))
    {
      err = GetLastError();
      UIWinPrintOSErr (gP->uiWinH, "GdbExecuteCmdLine: ",
          err, 0);
    }
  else
    err = 0;

  return err;
}
```

When making calls to functions with many parameters, insert comments for those parameter values whose meaning is not clear from the name of the argument passed.

For example:

```
  PrvWaitComplete (gP, false /*pingCheck*/, false /*retLinkOn*/,
        true /*echo*/, 0 /*resultsPP*/, 500 /*timeout*/);
```

Every `#endif` and `#else` should have a comment describing what it corresponds to:

```
#ifdef foo
  ...
#else    // not foo
 ...
#endif   // not foo

#ifdef foo
  ...
#endif   // foo

#ifndef foo
  ...
#else    // foo
  ...
#endif   // foo

#ifndef foo
  ...
#endif   // not foo
```

When minor changes are made to a source file that has been fairly stable for a while, put a comment like the following next to the change. This makes it easier to locate. The word `chg` should immediately follow the left angle bracket, the date should be day-month-year, and the author's initials should follow. This comment string can later be taken out of the source for better readability, after the change has proven to be stable.

```
// <chg 10-Dec-98 AB> Now check for nil pointer on entry
```

When you are aware of shortcomings in the code, or are putting a particular feature off until later, put a `DOLATER` string commenting what needs to be done:

```
// DOLATER need to optimize this to work better with long filenames....
```

2.6 Naming Conventions

For the most part, the Handspring naming conventions are the same as those used for Palm OS.

2.6.1 Capitalization

The general rule for naming functions, variables, and constants is to never use underscores. When a name contains multiple sub-words, each subsequent sub-word is capitalized:

```
// Function names, global variables, and typedefs are initial upper case:
int           AppFunctionName();
extern int    AppGlobalVar;
typedef int   AppInt;
typedef enum  {appModeVerbose, appModeQuiet} AppModeEnum;

// Local variables, constants, enums values, and structure members
//  are initial lower case:
int           myLocalVar;
#define       myBogusConstant        4
enum          { myEnum1, myEnum2 };
typedef struct
 {
   int myMember1;
   int myMember2;
 } MyCustomType;
```

As illustrated above, global variables, functions, and names of types always start with a capital letter. Local variables, constants, enums, and structure members always start with a lower case letter.

`#define's` that control build and compile time options are all capital letters with underscores between the words:

```
#define ERROR_CHECK_LEVEL  0
```

2.6.2 Choosing Names

In general, don't be terse when naming functions or variables. An appropriate name is, in most cases, preferable to inserting comments for explanations.

All exported/public functions, global variables, and constants should start with a short (2-4) character mnemonic for the module or manager to which that function belongs. The sub-words in the name should progress from **general** to **specific**.

For example:

```
MemInit ();
MemPtrNew ();
MemPtrFree ();
MemHandleNew ();
MemHandleFree ();

DWord MemDebugFlags;

#define    memPtrNewFlagZeroInit   0x01
#define    memPtrNewFlagHiHeap     0x02
```

All private (static) functions and definitions within a single source file should be named with a prefix of `Prv`, such as:

```
static void
PrvInitPrefs (int fromFile);
#define prvMyPrivateConstant    1

typedef struct
  {
    DWord  member1;
    DWord  member2;
  } PrvMyPrefsType;
```

Functions that must be shared between different source files within a particular module, but not exported to the outside world, cannot be declared static. They should be named with `Prv` following the module mnemonic:

```
MemPrvInitGlobals ();
MemPrvFreeGlobals ();
```

All typedef names for structure types should end with the word `Type`. Simple scalar types should not have `Type` on the end. All pointer types should end with `Ptr`. All enum types should end with the word `Enum`. When naming enum constants, prefix each of the constants with the name of the enum:

```
typedef struct
  {
    DWord member1;
    DWord member2;
  } MyPrefsType;
typedef Word  MyWord;
typedef Word* MyWordPr;
typedef enum
  {
    memOpenModeVerbose,
    memOpenModeQuiet
  } MemOpenModeEnum;
```

For local and global variables, append a `P` to the end of pointer variables and an `H` to the end of handle variables:

```
Byte*        myDataP;
Handle       myDataH;
```

2.6.3 Resource ID names

Nearly every Palm OS application has a header file called `MyAppRsc.h`. This header file defines the resource IDs and menu command IDs for the resources and menus of the application. Following the conventions described above, a typical `Rsc.h` header file should look like this:

```
// List View
#define rscListViewFormID                    1000
#define rscListViewCategoryTriggerID         1003
#define rscListViewCategoryListID            1004
#define rscListViewTableID                   1005
#define rscListViewLookupFieldID             1007
#define rscListViewNewButtonID               1008
#define rscListViewUpButtonID                1009
#define rscListViewDownButtonID              1010
// Details Dialog Box
#define rscDetailsDialogFormID               1200
#define rscDetailsDialogCategoryTriggerID    1204
#define rscDetailsDialogCategoryListID       1205
#define rscDetailsDialogSecretCheckboxID     1207
#define rscDetailsDialogOKButtonID           1208
```

```
#define rscDetailsDialogCancelButtonID        1209
#define rscDetailsDialogDeleteButtonID        1210
#define rscDetailsDialogNoteButtonID          1211
#define rscDetailsDialogPhoneListID           1213
#define rscDetailsDialogPhoneTriggerID        1214
// Menu Bars
#define rscListViewMenuBarID                  1000
#define rscRecordViewMenuBarID                1100
#define rscEditViewMenuBarID                  1200

// Menu commands
#define rscListViewRecordSendCategoryCmd      100
#define rscListViewRecordSendBusinessCardCmd  101
// Delete Note Alert
#define rscDeleteNoteAlertID                  2001
#define rscDeleteNoteAlertYesButtonIndex      0
#define rscDeleteNoteAlertNoButtonIndex       1
```

Note that all the defines start with `rsc`. The next part of the name should contain the form or dialog name, like `ListView`, or `DetailsDialog`. The next part of the name should describe the area of the form, if applicable, such as `Category`. The last part of the name should contain the object type followed by `ID`, such as `ButtonID`, or `FieldID`.

Menu commands are named similarly to form object IDs, except that they are followed by `Cmd` instead of `ID`. Likewise, dialog button indices end with `Index`.

2.6.4 Filenames

Filenames should be named following the same conventions as used for functions: initial capital letter, sub-words capitalized, no underscores, and sub-words progressing from general to specific:

```
MemEntry.c
MemHeapUtils.c
MemPtrUtils.c
MemMgr.h
```

Private *include* files for a particular module or manager should have `Prv` on the end of them:

```
MemPrv.h
SystemPrv.h
```

2.7 Basic Types

In order to make code as portable as possible, avoid the use of the standard C types, such as int, long, short, etc. These types are different sizes, depending upon the different platforms and compilers. Instead, use the basic type names below that are defined in the header file <*Common.h*> (<*PalmTypes.h*> in Palm OS 3.5). This header file contains `#ifdefs` to guarantee that the basic types below are the same size on all platforms and compilers.

In general, function parameters should always be declared using fixed size types so that they remain portable even when different functions within a system are compiled using different compilers. The variable size types below (Int, UInt, etc.) should only be used for local variables. Avoid declaring function parameters using enums, since the size of an enum can vary from compiler to compiler.

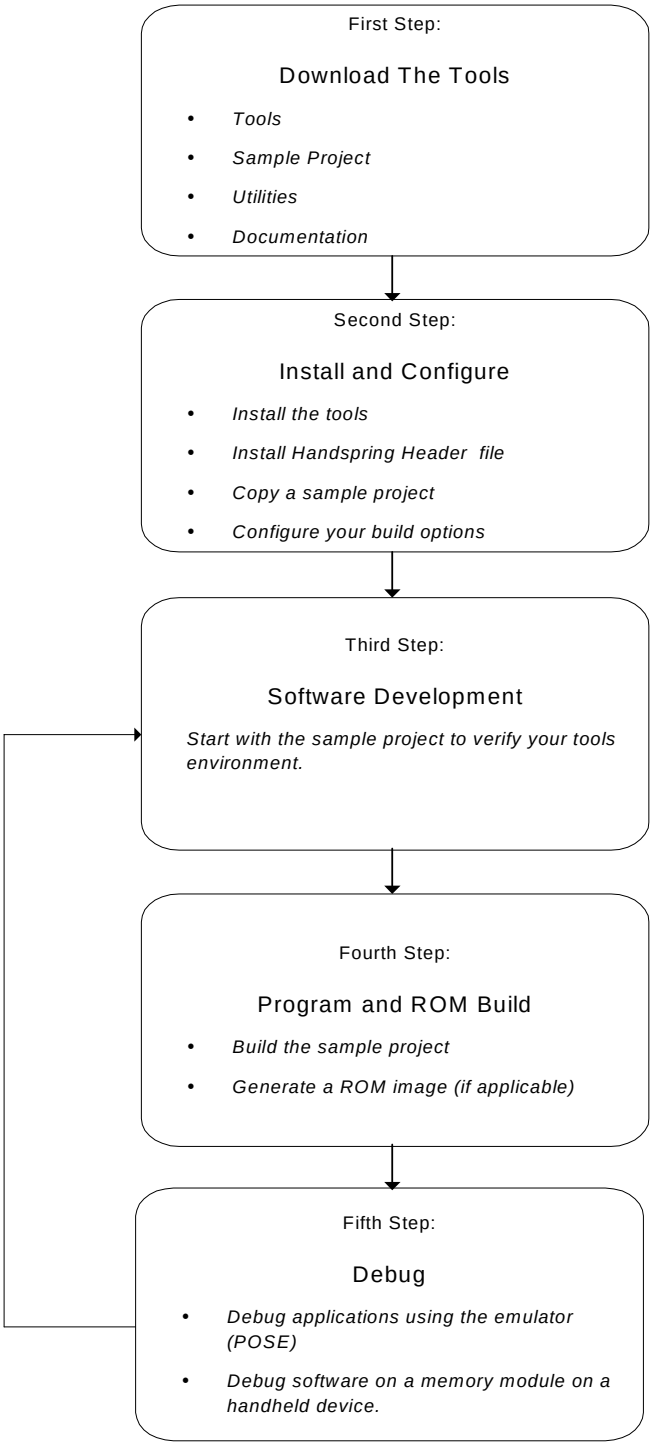
Table 1. Field Types

Format		Size	Sign	Use...
Palm OS 3.1	***Palm OS 3.5***			
Byte	UInt8	8 bits	unsigned	...when you need 8 bit quantities
SByte	Int8	8 bits	signed	...when you need 8 bit quantities
Word	UInt16	16 bits	unsigned	...when you need 16 bit quantities
SWord	Int16	16 bits	signed	...when you need 16 bit quantities
DWord	Int32	32 bits	unsigned	...when you need 32 bit quantities
SDWord	Int32	32 bits	signed	...when you need 32 bit quantities
UChar	Uchar	8 bits	unsigned	...with unsigned 8 bit character arrays
Char	Char	8 bits	signed	...with signed 8 bit character arrays
Boolean	Boolean	8 bits	unsigned	...when you need on/off boolean quantities
UInt	UInt	at least 16 bits	unsigned	...only for local variables, never for function parameters
Int	Int	at least 16 bits	signed	...only for local variables, never for function parameters
ULong		at least 32 bits	unsigned	...only for local variables, never for function parameters
Long		at least 32 bits	signed	...only for local variables, never for function parameters

3 Handspring Palm OS GNU Tools – Getting Started

The following diagram shows the general process of getting started with Handspring's development tools. We'll cover each of these steps in sequence to get you developing quickly.

Figure 1. Getting Started with Development Tools



3.1 First Step: Download The Tools

Everything that you need to get started can be downloaded from the Handspring website.

3.1.1 If You Are New to GNU C Development

It is worthwhile to understand the general GNU C development environment and tools before diving into the nuances of Springboard development. The on-line documentation listed in the *GNU References* chapter may be helpful. In addition, third-party books are available covering all varieties of UNIX-based development tools (e.g., O'Reilly & Associates). You may find it particularly useful to review documentation covering the Bash shell, GNU C compiler, and make utility.

Go to: http://www.oreilly.com/

3.1.2 Development Environment and Tools

Go to: http://www.handspring.com/developers/sw_dev.jhtml

Download *PalmOSGNUToolsWithCygwin*. This is Handspring's Palm OS GNU Tools with Cygwin GNU utilities for Windows.

Download *HandspringHeaders.exe*. These header files are required to develop Springboard module applications with plug-and-play capability.

3.1.3 Sample Projects

Go to: http://www.handspring.com/developers/sw_dev.jhtml

- Download *Tex2Hex*. This is a generic GNU project for Palm OS applications. This is a generic Palm OS application with no Handspring specific calls.
- Download *DiagRefModule* (optional). This is a reference for Springboard communications modules. This sample project includes a custom serial library and simple terminal program that can be used with a Springboard module containing a UART, Flash memory, and other common components. Full details of this module can be found on the Application Note entitled *Diagnostic Reference Module* on the Handspring website.
- This may also be a good point to obtain Springboard hardware for development purposes. A 8MB Flash module can be used to test software that will reside on a module. A Diagnostic Module can be useful when examining source code that implement Springboard specific features.

Go to: http://www.handspring.com/developers/tech_notes.jhtml

3.1.4 Utilities

Go to: http://www.handspring.com/developers/sw_dev.jhtml

- Download DebugPrefs. This is a useful utility that enables you to easily enter debug and console modes on a Handspring handheld either through the serial or USB ports. It also includes features that help trap Fatal Exception errors.
- Download Palm-Debugger. This is an updated debugger that works with Windows 2000.

3.1.5 General Documentation

Handspring's Development Guides contain detailed hardware and software documentation on the different Handspring products including the Springboard Expansion Slot. The manuals can be found on our web site at:

http://www.handspring.com/developers/documentation.jhtml

Developing software for Handspring handhelds is the same as developing for other Palm OS-based systems. The Palm web site (http://www.palmos.com/dev/support/docs/) contains a lot of information covering standard Palm OS development. Some specific reading includes:

- *Palm OS SDK Reference*

- *Palm OS Programmer's Companion*
- *Development Tools Guide*
- Recommendations on third-party books.

This site also contains specific documentation regarding Palm OS releases that may be helpful. These are very important to follow to maintain compatibility across difference devices.

- *Making Your Application Run on Different Devices* discusses writing software that works across Palm OS releases. This section is located in the *Good Design Practices* chapter in the *Palm OS Programmer's Companion*.
- *Compatibility Guide* covers differences between various Palm OS releases. Handspring Visor is based on Palm OS 3.1. This section is located in the *Palm OS SDK Reference*.

3.2 Second Step: Install and Configure

3.2.1 Installation

- Install the development tools that you downloaded.
- Install the Handspring header file.
- Create a project directory and copy the sample *Tex2Hex* project there. After you have installed the tools, you will find an area for sample projects in a directory named `/PalmDev/Samples`

3.2.2 Configuration

- The installation process creates all the necessary mount points, paths, and other configuration details; however, to review how your system is setup, the details of the installation are covered in the next chapter (Handspring Palm OS GNU Tools, *Manual Installation* chapter).
- Set your build options. These are standard GNU GCC options. In the SDKs directory, there is a subdirectory called `BuildOptions\Build`. Using the Cygwin shell, run the makefile in this directory with the appropriate options.

  ```
  >make <option>
  ```

optsDev	-	set std development options (affects all makefiles)
optsTest	-	set std test options (affects all makefiles)
optsRel	-	set std release options (affects all makefiles)

- Configuring your tools to work with Microsoft Developer Studio is covered in the next chapter.

3.3 Third Step: Software Development

Here are some tips that should make development in the Handspring environment easier.

- Handspring extensions are defined in *HsExt.h* and are documented in the Springboard Development Guide.
- Review Handspring Coding Standards. All sample code from Handspring will follow these conventions.
- C source files are created and edited in the `/Src` directory of your project.
- Resource files that describe Palm OS resources -- like bitmaps and forms -- are created and edited in the `/Rsc` directory of your project. The resources are compiled by Palm-RC (as described later in this guide). The resource types are described in Palm OS SDK documentation. In particular the *Palm OS Companion Guide* covers different Palm OS resource types.
- The project makefile is created and edited in the `/Build` directory. The compiler, linker, Palm-RC, and other tools (as appropriate) are called from this makefile. BASH scripting and make commands can be found later in this guide. As appropriate, Palm-MakeROM is called here to create a ROM image. An example of this can be found in the `DiagRefModule` sample project.

3.4 Fourth Step: Program and ROM Build

3.4.1 Build Process

- Enter the Bash shell and execute the makefile in your sample project's `/Build` directory.
- Intermediate files are stored in the `/Obj` directory and are kept or deleted depending on your BuildOptions setting.
- Symbol files for debugging are generated in the `/Obj` directory.
- The executable *.prc* file is copied from the `/Obj` directory to the `/prc-tools/bin/Device` directory.

3.4.2 Generate a ROM Image

- A sample of this process is in the *DiagRefModule* sample project. The Palm-MakeROM utility is called from the project makefile in the `/Build` directory.
- Palm-MakeROM takes parameters specified in the makefile such as ROM access time and *.prc* files to include, and generates a .bin file. This .bin file can be programmed onto a Flash module ROM (like the Handspring 8MB Flash Module) through either the Palm-Debugger, or an end-user oriented utility called CardUpdaterMaker. The Palm-MakeROM reference is included in this guide.

3.4.3 Creating Flash ROM Updates for Customers

- Customers need a way to upgrade developer's modules in the field. To meet this requirement, Handspring has created an SDK called *CardUpdaterMaker*. This can be found on our website at:

  http://www.handspring.com/developers/sw_dev.jhtml

- The *CardUpdaterMaker* SDK is designed to call your project makefile to build the latest version of your prcs.
- The *.bin* file created by your project makefile is then used and incorporated into an Updater *.prc* that CardUpdaterMaker builds.
- The final Updater *.prc* is placed in the `/prc-tools/bin/Device` directory.
- Customers can use this program to easily update the Flash ROM built into your module.

3.5 Fifth Step: Debugging

3.5.1 Debugging on a Handheld

- Once the `.prc` is made, it can be copied to the handheld through the Install Tool and Hotsync, or via the Palm-Debugger.
- On the handheld, the DebugPrefs utility can be used to force Console and Debug modes to use either the serial or USB port. This will aid in debugging your application.
- References to the full Palm-Debugger application can be found later in this guide.

3.5.2 Debugging on the Emulator (POSE)

POSE can be used to speed up the debugging process. Once the *.prc* is made, you can drag-and-drop it onto the Palm OS Emulator (POSE). Note that ROM images are *not* included with POSE. There are several ways to obtain the appropriate ROM files for the POSE that will emulate Handspring hardware. These methods are referenced on Handspring's website:

http://www.handspring.com/developers/tech_pose.jhtml

Full documentation on the POSE can be found on the Palm website.

4 Handspring Palm OS GNU Tools

4.1 About the Handspring Palm OS GNU Tools

The Handspring Palm OS GNU (HPG) tools is a complete set of development tools for creating and debugging applications for Handspring Palm OS devices. As the name implies, these tools are based on the Free Software Foundation's GNU tools. The HPG tools include a C compiler, a resource compiler and *..prc* builder and a source-level debugger for debugging *applications*, *extensions*, or *libraries*.

The source code for the GNU tools is freely available on the Internet under terms of the GNU General Public License, as are all derivatives based on the GNU source code, including the HPG tools.

The GNU source code has been ported to many different platforms and operating systems and, in general, the HPG tools can be easily ported to any platform as well. Internally, though, these tools are only tested and used extensively on Windows NT/2000 and Windows 98 on x86 PCs, so your “mileage may vary” if you use them on a different platform or OS.

The HPG tools were originally developed starting from the prc-tools-0.5.0 distribution provided by Jeff Dionne. Since then, they have been upgraded to use the prc-tools 2.0 toolset maintained by Palm, Inc. Handspring’s approach is to build and run the HPG tools using Cygnus's cygwin32 running on Windows NT/2000 and Windows 98. Cygnus provides a set of GNU executables, including a Bash shell, make, sed, etc., that run on top of Windows and use the Windows file system. This solution is ideal because it provides all the power and flexibility of a Unix-like environment -- including powerful make and scripting abilities -- as well as access to traditional Windows applications all on one machine.

In addition to the traditional GNU command line tools, the HPG toolset also provides a Windows version of the Palm OS Debugger. This application provides multiple windows for debugging a Palm OS device, and features:

- A window for assembly-level debugging.
- A window for shell-like functions (such as getting a directory listing of databases on the Palm OS device, and getting heap dumps).
- A window for source-level debugging.
- A simple scripting environment.
- Source-level debugging support (reads symbol files generated by the Palm OS gcc compiler.)

The PalmDebugger application can "talk" to a real Palm OS device over a serial port or USB. It can also talk to a virtual Palm OS device running as the Palm OS Emulator Windows application over TCP/IP. There is a Communications menu in PalmDebugger that allows you to select a way to talk to the Palm OS device.

4.2 Installation

To install the tools, simply run the supplied installer. This will install the Cygwin tools (if necessary) and the Handspring tools. If the installer detects that the Cygwin tools are not already installed on your system, it will enable that checkbox by default.

4.2.1 Manual Installation Of Tools

If you're curious, the following instructions describe how to copy the files and make the necessary batch file and registry file changes manually. All of the following are performed automatically if you run the installer application:

1. Install the Beta20 release of the Cygwin tools.

2. After installing the Cygwin tools, you will need to define the Unix paths to each of your hard drive partitions, and you'll need to define the Unix path */prc-tools* to point to the directory that will point to your prc-tools directory. This information is saved in the registry by the `"mount"` command, so it must be done only once. To create the paths, launch the Cygwin Bash shell from the Start menu and enter the following at the command prompt, modifying appropriately for your particular setup:

   ```
   mount C: /           # This makes C: your root drive
   mount D: /d          # If you have a D: drive...
   mount E: /e          # If you have an E: drive...
   mount F: /f          # If you have an F: drive...
   mount E:\\prc-tools /prc-tools
                        # If your prc-tools directory will
                        # be at E:\prc-tools
   ```

   If you make a mistake with the `mount` command, use the `"umount"` command to unmount a path. For example:

   ```
   mount F: /usr/local
   umount /usr/local
   mount E: /usr/local
   ```

   If you make a mistake when mapping the root drive, do the `umount` and the `mount` in one line, using a semi-colon:

   ```
   mount D: /
   umount /; mount C: /
   ```

   Note that if you include a backslash in the DOS path of your `mount` command, you need to use two backslashes (e.g., `'mount C:\\bin /bin'`).

3. Next, follow the instructions below for your particular operating system to ensure that the Cygwin tool executables and the *.prc*-tools executables are in your PATH setting:

   Windows NT/2000:

   In Windows NT and Windows 2000, go to the Properties panel of the System control panel applet and make sure the path to the *prc-tools\bin* directory is included in your path variable and make sure the path to the Cygwin tools is included as well. Here's an example:

   ```
   PATH=E:\prc-tools\bin;C:\cygnus\CYGWIN~1\H-I586~1\bin;%OLDPATH%
   ```

   Note that you can use the syntax `%VARNAME%` when defining the value of an environment variable in the Properties panel. In the above example, the previous path setting was copied into a variable named `OLDPATH`, and the PATH variable was defined in terms of `%OLDPATH%`.

You also need to define a few environment variables in order to be able to use the Visual Studio IDE to create a browser database and browse through Palm OS project source files:

```
PRCTOOLSDIR=E:\prc-tools
PALMDEVDIR=E:\prc-tools\PalmDev
PALMOSINC=%PALMDEVDIR%\sdk-3.5\include
HSINCDIR=E:\prc-tools\PalmDev\include\Handspring
```

Optionally, you can also add a HOME variable which will be used by the Bash shell when you type in cd without any arguments or when you use the ~ character in the Bash shell as a shortcut to your home directory. This variable must be defined as a Unix-style path. For example:

```
HOME=/e/projects
```

In Bash, to go to this directory type:

```
> CD ~
```

Windows 95/98:

In Windows 95/98, edit your *C:\Autoexec.bat* file to ensure that the path to the *prc-tools\bin* directory is included in your path variable. Make sure the path to the Cygwin tools is included as well.

You also need to define a few environment variables in order to be able to use the Visual Studio IDE to create a browser database and browse through Palm OS projects.

Optionally, you can add a HOME variable, which will be used by the Bash shell when you type in cd without any arguments, or when you use the ~ character in the Bash shell as a shortcut to your home directory. This variable must be defined as a Unix-style path. For example:

```
HOME=/e/projects
```

In Bash, to go to this directory type:

```
> CD ~
```

You should also make a call to the Visual Studio batch file that sets up environment variables for command line Visual Studio tools as well.

Here is a sample *autoexec.bat* file. This assumes that you have copied the *VCVars32.bat* file from the Visual Studio directory to the root directory of your C: drive.

```
SET PATH=c:\CYGNUS\CYGWIN~1\H-I586~1\BIN;%PATH%
SET PATH=E:\prc-tools\bin;%PATH%
SET HOME=/e/projects
SET PRCTOOLSDIR=E:\prc-tools
SET PALMDEVDIR=E:\prc-tools\PalmDev
SET PALMOSINC=%PALMDEVDIR%\sdk-3.5\include
SET HSINCDIR=E:\prc-tools\PalmDev\include\Handspring

CALL VCVars32.bat
```

Finally, you will need to add the following line to your *C:\Config.sys* file. This is necessary in order to launch a build using the GNU tools directly from the Visual Studio IDE, because the default variable environment space in the Windows 95/98 DOS shell is too small:

```
SHELL=C:\COMMAND.COM /p /e:4096
```

4. Copy the entire prc-tools directory to your hard drive to the location you chose in step 2. In this example, the prc-tools directory is copied to the root level of the E: drive.

Alternately, if you would rather re-build all of the HPG tools from the sources, launch the Cygwin Bash shell from the **Start** menu, and build the Palm tools using the following commands, adjust the `cd` command argument to where you installed the sources. This makefile for the tools will always create the directory /prc-tools and place the built set of tools into that directory.

```
cd /e/projects/tools/build
make
```

5. Make a shortcut to the *Palm-Bash.bat* file found in the */prc-tools/bin* directory and save it to your desktop (or some other convenient place). This shortcut opens up a DOS window and runs the Bash shell in it. This window should be used to execute any of the command line tools (e.g., make, m68k-palmos-gcc) because it sets up all additional required environment variables correctly for you (besides the static ones you already setup in step 3).

6. In order to be able to launch the GNU make tool directly from Visual Studio, you will also need to add the paths to the Cygwin and *.prc*-tools executables to Visual Studio's preferences. To do this:

    a. Launch Visual Studio and select **Tools > Options**.

    b. Select the **Directories** tab

    c. Select **Executable files** from the **Show** directory's **for:** pop up list.

    d. Add the following directories to the list (note that this line may be slightly different on different systems):

        - *C:\Cygnus\B19\H-i386-cygwin32\bin*
        - *E:\prc-tools\bin*

4.2.2 Using the Function Pop-up Developer Studio add-in

A Microsoft Developer Studio add-in is also provided with the HPG tools. This provides a convenient "function pop-up" window that is very useful during editing in order to go to a particular function in a source file. Unlike the built-in browser functionality that provides a similar capability, the function pop-up works before the code is compiled, and can display functions either in alphabetical order (if the [Shift] key is held down) or the order in which they appear in the source file.

To configure Developer Studio to use the function pop-up, do the following:

1. Choose the **Add-ins and Macro Files** tab from the **Tools > Customize** menu item dialog.

2. Click the **Browse** button and select the *FunctionPopUp.dll* file from the *prc-tools\bin* directory into which the installer installed the tools. You will need to change the **Files of type** option to **Add-ins (.dll)** in order to browse for DLL files.

3. This creates a toolbar icon on your screen that will activate the function pop-up for the top-most editor window. If you would also like to set up a keyboard shortcut for this command, do the following:

    a. Switch to the **Keyboard** tab of the same dialog

    b. Change the **Category** to **Add-ins** and select **FunctionPopUpCommand** from the **Commands** field.

    c. Put the cursor in the **Press new shortcut key** field, click **Ctrl+,** then click the **Assign** button.

d. Put the cursor into the **Press new shortcut key** field, click **Ctrl+Shift+;** then press the **Assign** button.

Because the function pop-up displays functions in alphabetical order when the shift key is held down, these assignments will cause Ctrl+ to bring up the function pop-up in file order and Ctrl+Shift+ to bring it up in alphabetical order.

4.3 Overview of Available Tools

4.3.1 Palm-Specific tools

Used for generic Palm OS programming on the Handspring and Palm product lines.

Palm-Debugger: Assembly- and source-level debugger and console window for debugging an actual Palm OS device over the serial or USB port, or through TCP/IP to the Palm OS Emulator.

Palm-OSEmulator: Emulates a Palm OS device as an application on the desktop. This is a desktop (Windows, Macintosh, and Linux) application that emulates a Palm OS device.

Palm-MakeROM: Creates a ROM file image from a set of *.prc* files. Can also be used to print information on an existing ROM image, break down an existing ROM image into a set of *.prc* files, or patch an existing ROM image by adding ROM tokens.

Palm-RC: Utility for creating a *.prc* file by combining data from one or more of the following: text resource description files, *.rsrc* binary resource files created by Metrowerks Constructor or ResEdit, or linked object code output from the m68k-palmos-gcc compiler/linker. This utility can also convert most of the resources in an existing *.prc* file back into text resource description files.

Palm-PrcDump: Utility for dumping the contents of a *.prc* file in hex for examination and debugging purposes.

Palm-Pretty: Utility for re-formatting source code to conform to the Handspring coding conventions. This is a Bash script file that passes the appropriate command line options to the GNU indent program. Type `Palm-Pretty` by itself on the command line for usage information.

Palm-RunGnu.bat: Batch file that can be called from the Visual Studio IDE or from a DOS prompt to run a command line GNU tool (such as "make") with all appropriate environment variables set up correctly for the Palm-GNU tools.

Palm-Bash.bat: Batch file that will launch the GNU Bash shell for interactive use with all the correct environment variables set up for using the Palm-GNU tools. The batch file uses *Palm-RunGnu* to launch the Bash shell in interactive mode.

Note: Most of the tools have built-in help available when they are invoked with the '--help' command-line option.

4.3.2 General Purpose GNU tools

These tools are documented in the GNU References, later in this guide.

bash - GNU command line shell with scripting support, filename expansion, etc.

make - GNU make utility.

indent - GNU program for reformatting source code.

4.3.3 GNU tools for Palm OS development

These tools are documented in the GNU References, later in this guide.

m68k-palmos-gcc: C compiler, assembler, and linker.

m68k-palmos-gdb: Command line-based source-level debugger.

m68k-palmos-objdump: Object code dumper and disassembler. Can also be used to dump symbol information contained in object files or final executables.

4.4 Pitfalls to Avoid!

4.4.1 make

There is a bug in the VPATH treatment of *make* when running in Windows 98 (and likely 95 as well). When using VPATH, *make* is case-sensitive, whereas normally under Windows it is not. If you've checked out the sources to the GNU tools (binutils, for example) under Windows NT and then switch to Windows 95, the names of the files that fit in 8.3 will appear as all uppercase in a directory listing and **won't** be recognized by *make* when it uses VPATH.

In order to run *Bash* in Windows95/98, you need to increase the size of the environment space to 4096 bytes. Otherwise, you will get an `out of environment space` error. You can do this from the Properties panel for the cygwin shortcut or by launching command.com with the /e:4096 command line option.

4.4.2 m68k-palmos-gcc

When compiling large applications (>32K of code), you may see the following error message:

```
Tex2Hex.s: Assembler messages:
Tex2Hex.s:84: Error: Signed .word overflow; switch may be too large;
42146 at 0x50ca
```

This is usually the result of the compiler attempting to reference a pre-initialized global variable. To determine the "real" line that is causing the problem, pass the `-save-temps` option to the compiler and look at the output *.s* file that is generated.

This problem is due to the compiler using intermediate 16-bit numbers to represent pre-initialized global variable references in the code. These 16-bit offsets start at higher numbers based on the amount of code you have. Unfortunately, there doesn't seem to be any workaround for this except for not using pre-initialized globals in the first place. You can either declare them uninitialized and then initialize them manually in at the start of `PilotMain()`, or if they are never changed, put the `const` keyword in front of them so that they end up in the code segment.

4.5 Using the Tools

This section provides an overview of how to use the HPG tools for developing and debugging a Palm OS application. The HPG tools can be used to compile and build Palm OS applications either directly from the command line or from an Integrated Development Environment (IDE) such as Microsoft's Visual Studio. The following examples illustrate how to use the tools with Visual Studio.

Besides the obvious benefits of collecting all your source files into a single project window, the IDE also provides source code browsing support. With the appropriate Visual Studio project file, you can use the IDE's built-in browser functionality for locating function prototypes, type and macro definitions, and function implementations for your application. You can also run the makefile for your application directly from the IDE and view the output from the make utility in the IDE's output window.

The general steps for creating, building, and debugging a Palm OS application are as follows:

1. Create a makefile and Visual Studio project for your application. This is typically done by copying a pre-existing project, such as the *Tex2Hex* sample.

2. Create *.h* and *.c* source files for your application code.

3. Create a *Palm-RC* *.rcp* text file describing the resources in your application.

4. Build your application using the **Build** menu in the IDE. This does two things: it fires off the GNU *make* utility which runs through your makefile to build your Palm OS application. It also compiles all the

sources using the Visual Studio compiler in order to generate a browser database. The x86 object code output from this compiler is not used, of course.

5. If you haven't launched the console support on the Palm OS device yet, do so now by entering `Shortcut - . - 2` using Graffiti (that's the shortcut stroke, followed by two taps to generate a period, followed by the number 2).

6. Use the PalmDebugger application to load your application onto the device and debug it. PalmDebugger provides both assembly and source-level debugging (using a symbol table generated by the m68k-palmos-gcc compiler) as well as a "console" shell window.

7. Switch back to the Visual Studio IDE, edit your source code as needed, then repeat from step 4.

5 GNU References

The GNU Toolkit upon which the Handspring Palm OS GNU Tools are based, is well documented. Here are links to the full documentation for reference purposes.

Main GNU Web Page:

http://www.gnu.org

GNU Miscellaneous Documentation:

http://www.gnu.org/doc/doc.html

GNU Manuals Online:

http://www.gnu.org/manual/manual.html

GNU Docs in Japanese:

http://www.gnu.org/software/gnujdoc/gnujdoc.html

Make Overview:

http://www.gnu.org/software/make/make.html

GNU Make Manual:

http://www.gnu.org/manual/make-3.79.1/html_mono/make.html

6 Palm-MakeROM Overview

6.1 Description

Of all the utilities in the Handspring GNU Tools package, *Palm-MakeROM* is one of the most important and most used tools for Springboard Module development. This tool is responsible for generating the ROM image file that will be flashed onto the Module's Flash memory. The utility will take in standard Palm OS *.prc* files and assemble them into the appropriate format for the *.bin* file. The *.bin* file will be in Motorola (big-endian) byte order. The utility will also set the appropriate header information for the ROM module (as specified on the command line). In addition to generating the Flash ROM image files, the utility can also be used to print info on an existing ROM image, break down an existing ROM image into a set of *prc* files, or patch an existing ROM image by adding ROM tokens.

6.2 Usage Summary

```
Palm-MakeROM --help
Palm-MakeROM <lots `o options…see below>
```

General Options	
`-h, --help`	Print this help message.
`-o <outFile>`	Specify the filename, `outfile`, for the *.bin* file.
`-noSpaces`	Replace spaces by underscores (`_`) in *.prc*filenames when doing `-op break`.
`-noForceReadOnly`	Do not force each *prc* to be marked as read-only. By default *Palm-MakeROM* will set each *prc* as read-only. Using this option will **not** force each *prc* in the ROM image to be marked as read-only.
`-copyPrevention`	Force the copy prevention (beam prevent) bit to be set in the *prc* files in the ROM image.
`-autoSize`	Automatically size the ROM image to fit the size of all the input files. The `-romBlock` option will be an optional maximum size parameter. When using this option, the `-chRomTokens` parameter is not needed. Use of this flag is recommended.
Modes of Operation (only 1 can be specified)	
`-op info <romName>`	Print info on the specified ROM image.
`-op patch <romName>`	Used when patching a previously created ROM image. This will specify the file to patch. **NOTE: When patching ROM images, this must be the first option on the command line.**
`-op join <smallROM> <bigROM> <bigROMOffset>`	Join Small and Big ROM image files into a single ROM image.
`-op split <romName> <bigROMOffset>`	Split `romName` into a Small and Big ROM image files with names of *outFile1.bin* and *outFile2.bin*.
`-op create`	Used to create a new ROM image. The *prc* files used for this ROM image are specified with the `-romDB` and `-romBootDB` options. The output ROM image file name is specified with the *-o* option.
`-op break <romName>`	Extract the *prc* files from a given ROM image. The resulting *prc* files will be written into the current working directory.

Card Header Parameters	
`-base <start>`	Base address of card specified in hexadecimal (example: 0x10000000).
`-hdr <cardHdrOffset>`	Offset from the base of the module memory to the module header. This must be 0x08000000 for all Handspring removable modules. This offset is added to the module's logical base address in Palm OS of 0x20000000. With this setup, the logical address of the module's ROM image is at 0x28000000.
`-chRomTokens <offset> <size>`	This option is not required if the –`autoSize` option is used. The offset and size of the ROM token area on the module. This area is used to store data specified in the –`tokStr` parameter. The *offset* should be set to the end of the ROM minus space for the tokens themselves. In the above example, the ROM offset is at 0x08000000 and its size is 0x10000 (64K), so the ROM tokens are put at 0x08000FF00 with a size of 0x100.
`-chBigRomOffset <offset>`	Offset to Big ROM from card base (example: 0x00C08000).
`-chName <name>`	The ASCII card name of the module as registered with Handspring Developer's web site. Must be in quotes and can be up to 31 characters in length. (example: "`HandspringCard`").
`-chManuf <name>`	The ASCII manufacturer name as registered with Handspring Developer's web site. Must be in quotes and can be up to 31 characters in length (example: "`Handspring, Inc`.").
`-chVersion <version>`	The 16-bit version number of the module. The developer determines the version number to be used in this parameter. A typical use is to store the major version in the high byte and the minor version in the low byte.
`-chStack <initStack>`	Initial stack pointer (example: 0x00003000).
`-chChecksum [<bytes>]`	Number of bytes to use for checksum calculation specified in hexadecimal. If no parameter is specified, use the entire ROM image for the checksum calculation (example: 0x00001000).
`-chReset0`	Reset vector should point to alias of card header at 0x00000000.
`-chZCrDate`	Set the creation date to 0. **Note: Used for testing purposes.**
ROM Store Options	
`-romName <name>`	Name of the ROM store on the module. This parameter is for descriptive purposes only. Must be in quotes and can be up to 31 characters in length (example: "`ROM Store`").
`-romBootDB <filename>`	Name of boot code *prc* file (boot=10000 becomes reset vector) (boot=10001 becomes initCodeOffset1) (boot=10002 becomes initCodeOffset2).

`-romHalDB <filename>`	Name of HAL *prc* file (boot=10000 becomes reset vector) and -romBootDB's reset vector gets moved into `initStack` field of card header.
`-romPalmHalDB <filename>`	Replaces -`romHalDB` option for Palm OS 3.5 (boot=19000 becomes `halDispatch` in CardHeader).
`-romDB <fileName>`	Name of *prc* file to put in ROM store. Must be in quotes (example: "*MyApp.prc*").
`-romBlock <offset> <size>`	The offset and size of the ROM area on the module relative to the module base address. The offset must be 0x08000000 for all Handspring modules. The size is the total size of the formatted ROM area used by the Palm-MakeROM tool, which can be less than or equal to the size of the ROM chip itself. This option is not required if the –`autoSize` option is used.
RAM Store Options	
`-ramBlock <offset> <size> ...`	Offset and size of RAM block on the card specified in hexadecimal (example: 0x0 0x0).
ROM Token Options	
`-tokStr <type> <str>`	The ID and value of a ROM token to be placed in the ROM token area specified by the –`chRomTokens` option. This option can be repeated for every ROM token that needs to be included. The ID must be a string of four characters. The value can be any number of characters long. All Handspring removable modules should contain an `'HsAT'` token with a value specifying the required access time of the chip selects in nanoseconds. For modules without this value, the chip select access time will be set to the slowest possible value allowed by the base unit hardware. For example, -`tokStr HsAT 200`, will set the access time to 200 ns. Optionally, you can also include an `'HsWR'` ROM token. The presence of this token (the value is ignored) tells the system to launch the module welcome application on the module if the module is inserted during a soft or hard reset. Normally, the module welcome application is not launched after a reset. (example: `-tokStr HsWR 1`.
`-tokHex <type> <str>`	Include hex data as inline token. (example`: -tokHex "snum" "FFAABBCC"`)
`-tokBin <type> <filename>`	Include binary file as pointer token.
`-tokPrc <type> <filename> <resType <resID>`	Include resource from *prc* file as pointer token.

6.2.1 Examples

A developer at Handspring, Inc. would like to place a new application on a 8MB Flash Module for testing purposes. The application consists of a single game application along with a module Welcome application that

will be used as a splash screen when the module is inserted. The developer has registered the module and company name with Handspring, and all the applications have been built with the appropriate creator and type codes. To make ROM size calculations easier, the developer will use the `-autoSize` option. The access time to the module memory is 200 ns and will be specified with the `HsAT` token.

NOTE: The command line options have been split into multiple lines for easier reading. When using the application, all the options must be on a single line.

```
> Palm-MakeROM -op create
              -hdr 0x08000000
              -chName "PatricksGames"
              -chManuf "Handspring, Inc."
              -chVersion 0x0100
              -romName "ROM Store"
              -autoSize
              -tokStr HsAT 200
              -romDB "WelcomeApp.prc"
              -romDB "Gungnir.prc"
              -o GamesROM.bin
```

After running the program, the output file *GamesROM.bin* will be in the current working directory. The developer can then use this ROM image to flash a module for testing. As more applications are added to the ROM module, the developer simply adds more `-romDB` arguments to the command-line options.

Another developer has received a ROM image from someone and would like to view the contents of this ROM image. Use the following options to inspect an existing ROM image.

```
> Palm-MakeROM -op info FansomeROM.bin
General Info:
  cardBase            : 0x10000000
  cardHdrOffset       : 0x08000000
  romBlockOffset      : 0x08000000
  romBlockSize        : 0x00100000

CardHeader Info:
  initStack           : 0x00003000
  resetVector         : 0x00000000
  hdrVersion          : 4
  flags               : 0x0020
  name                : BigRedModule
  manuf               : Handspring, Inc.
  version             : 0x0100
  creationDate        : 0xB60350A7
  numRAMBlocks        : 0
  blockListOffset     : 0x08000200
  readWriteParmsOffset: 0x00000000
```

```
  readWriteParmsSize  : 0x00000000
  readOnlyParmsOffset : 0x0800ff00
  bigROMOffset        : 0x00c08000
  checksumBytes       : 0x00001000
  checksumvalue       : 0x9541

ROMStore Info:
  version             : 1
  flags               : 0x0000
  name                : ROM Store
  creationDate        : 0x00000000
  backupDate          : 0x00000000
  heapListOffset      : 0x08000208
    heap 0 offset     : 0x08000212
  initCodeOffset1     : 0x00000000
  initCodeOffset2     : 0x00000000
  databaseDirID       : 0x080FFEEE

ROM Token info:
    token 0           : HsAT, 3 bytes

Command line options to build:
  -base 0x10000000
  -hdr 0x08000000
  -chName " BigRedModule"
  -chManuf " Handspring, Inc."
  -chVersion 0x0100
  -chStack 0x00003000
  -chRomTokens 0x0800FF00 0x000F0100
  -romName "ROM Store"
  -romBlock 0x08000000 0x00100000
  -ramBlock 0x00000000 0x00000000
  -romDB "CardWelcome.prc"
  -romDB "FooBarApp.prc"
```

Yet another developer has received a ROM image, and for debugging purposes needs to extract all the *prc* files from this ROM image.

```
> Palm-MakeROM –op break FunnyROM.bin
Writing out database: "CardSetupApp.prc"...
Writing out database: "CardWelcome.prc"...
Writing out database: "Jokes.prc"...
```

The current working directory will contain the *prc* files listed above.

6.3 Palm-RC

6.3.1 Description

Palm-RC is a resource compiler for PalmOS applications. The full manual for this utility is in the appendix of this manual. The main difference between the Palm-RC for Handspring devices and the previous versions of Palm-RC is 16-bit color support for Visor Prism. For more information on these items and other options for Palm-RC, please consult Chapter 7, *PalmRC User Manual*.

6.4 HsSplit

6.4.1 Description

Splits the source file into a number of files of *chunkSize* bytes. The last output file may be smaller than *chunkSize*. The output filenames begin with the source file name, and are followed by ".nnnn", where *nnnn* is a four digit decimal number in the range 0000 - 9999. The source file is preserved.

6.4.2 Usage Summary

```
HsSplit --help
HsSplit -src <path> -chunkSize <# of bytes> -outDir <path> [-clobber]
```

Command Line Parameter	Usage
`--help`	Display the usage screen
`-src <path>`	Source file
`-chunkSize <# of bytes>`	Size of each output file in bytes
`-outDir <path>`	Directory for the output files
`-clobber`	If this option is specified, overwrite the existing files. Otherwise do not overwrite.

6.4.3 Examples

A user wants to split the file, *Handspring.bin* of size 570,418 bytes into 10,000 byte sized files. The output file should be in the current directory.

```
> HsSplit -src Handspring.bin –chunkSize 10000 –outDir .
Executing "hssplit -src Handspring.bin -chunkSize 10000 -outDir . ".
hssplit completed successfully: 58 files created (total size is 570418 bytes)
```

Resulting files range from *Handspring.bin.0000* to *Handspring.bin.0057* each of size 10,000 bytes except for *Handspring.bin.0057*, which is 418 bytes.

6.5 Palm-PrcDump

6.5.1 Description

A utility that dumps the contents of a Palm OS *prc* file to the screen. This utility displays the header information in an easy-to-read format and all the resource data in hex and ASCII values. This utility is useful for checking if various header information was compiled incorrectly, such as: name, type, creator, version, or attributes. If you are truly adventurous, you can use the hex data of the various resources to aid in debugging.

6.5.2 Usage Summary

```
Palm-PrcDump --help
Palm-PrcDump [-x] [-noData] <prcfile>
```

Command Line Parameter	Usage
--help	Display the usage screen
<prcfile>	The Palm OS *prc* file to dump
-x	Print the raw data in hex without any special formatting
-noData	Do not print the resource/record data. Useful for getting a quick summary of the *prc* file without looking at the data.

6.5.3 Examples

A user needs to inspect the Tex2Hex application to verify that the creator code and the name were set properly during compile time. The user does not care about the actual data in the resource.

```
> Palm-PrcDump -noData Tex2HexApp.prc
Name                  = 'Text to Hex'
Attributes            = 0x0001
   resourceDB            : on
   readOnly              : off
   appInfoDirty          : off
   backup                : off
   okToInstallNewer      : off
   resetAfterInstall     : off
   open                  : off
version               = 1
creationDate          = 0xB5BEFC22
modificationDate      = 0xB5BEFC22
lastBackupDate        = 0x00000000
modificationNumber    = 0
appInfoID             = 0x00000000
sortInfoID            = 0x00000000
type                  = appl
creator               = TxHx
uniqueIDSeed          = 0x00000000
recordList.nextRecordList = 0x00000000
recordList.numRecords = 13
recordList.entries:
entry 0: 'code' #0 offset:0x000000D2
0000004E:  636F6465 00000000 00D2                code......
entry 1: 'data' #0 offset:0x000000EA
00000058:  64617461 00000000 00EA                data......
entry 2: 'pref' #0 offset:0x00000126
00000062:  70726566 00000000 0126                pref.....&
entry 3: 'rloc' #0 offset:0x00000130
0000006C:  726C6F63 00000000 0130                rloc.....0
entry 4: 'code' #1 offset:0x00000132
00000076:  636F6465 00010000 0132                code.....2
entry 5: 'tFRM' #1000 offset:0x000007E2
00000080:  7446524D 03E80000 07E2                tFRM......
entry 6: 'tver' #1 offset:0x000008D2
0000008A:  74766572 00010000 08D2                tver......
entry 7: 'tAIB' #1000 offset:0x000008D6
00000094:  74414942 03E80000 08D6                tAIB.....╓
entry 8: 'Tbmp' #1000 offset:0x00000933
0000009E:  54626D70 03E80000 0933                Tbmp.....3
entry 9: 'Tbmp' #1001 offset:0x000009F6
000000A8:  54626D70 03E90000 09F6                Tbmp......
entry 10: 'MBAR' #1000 offset:0x00000B15
000000B2:  4D424152 03E80000 0B15                MBAR......
entry 11: 'Talt' #1000 offset:0x00000BC0
000000BC:  54616C74 03E80000 0BC0                Talt.....└
entry 12: 'Talt' #1001 offset:0x00000BFE
```

```
000000C6:  54616C74 03E90000 0BFE                Talt.....■

Resource: 'code' #0, 24 (0x18) bytes

Resource: 'data' #0, 60 (0x3C) bytes

Resource: 'pref' #0, 10 (0xA) bytes

Resource: 'rloc' #0, 2 (0x2) bytes

Resource: 'code' #1, 1712 (0x6B0) bytes

Resource: 'tFRM' #1000, 240 (0xF0) bytes

Resource: 'tver' #1, 4 (0x4) bytes

Resource: 'tAIB' #1000, 93 (0x5D) bytes

Resource: 'Tbmp' #1000, 195 (0xC3) bytes

Resource: 'Tbmp' #1001, 287 (0x11F) bytes

Resource: 'MBAR' #1000, 171 (0xAB) bytes

Resource: 'Talt' #1000, 62 (0x3E) bytes

Resource: 'Talt' #1001, 48 (0x30) bytes
```

Notice that the name and creator (boxed items in above output) are set to '`Text to Hex`' and `TxHx`, respectively. Notice that all the resources in the *prc* file were listed as well. If the `-noData` option were removed, you would see the hex data associated for each resource.

6.6 ToDos, ToMac, ToWin, ToUnix

6.6.1 Description

Utility to convert text files from one format to another. The source and destination formats it supports are DOS (*ToDos*), Unix (*ToUnix*), and Macintosh (*ToMac*).

6.6.2 Usage Summary

```
Text file newline conversion utility
Usage: ToXXX [--help] <file>...
```

Command Line Parameter	Usage
`--help`	Display the usage screen
`<file>`	File to convert. Program uses the same file as the output.

6.6.3 Examples

A user needs to convert the file, *bleh.txt* to Unix format.

```
> ToUnix.exe bleh.txt

Converting bleh.txt...
```

bleh.txt is now a Unix-formatted file, with the proper linefeed and carriage returns.

7 PalmRC User Manual

Palm-RC is based on PilRC written by Wes Cherry (wesc@ricochet.net)

7.1 Description

Palm-RC	A resource compiler/de-compiler and *.prc* builder for the Palm Pilot

7.2 Table of Contents

7.3 Usage

```
Palm-RC [<options>...]
```

Input file options:

`-rcp <infile>`	Input .RCP text file of resource descriptions (Use '-' for stdin).
`-rsrc <infile>`	Input Macintosh format resource file as generated by Metrowerks' Constructor or Apple's ResEdit.
`-gccApp <infile>`	Input application built with the Palm-gcc compiler/linker. This will create code 0, code 1, data 0, rloc 0, and pref 0 resources from the output of the Palm-gcc linker.
`-gccCode <infile> <resType> <resID>`	Input file built with the Palm-gcc compiler/linker as PalmOS code resource: `<resType>=<resID>`. Will also insure that `<infile>` was not built to use any globals.
`-gccCodeG <infile> <resType> <resID>`	Input file built with the Palm-gcc compiler/linker as PalmOS code resource: `<resType>=<resID>`. Unlike `-gccCode`, this option allows the code to use globals. This option is used to build "poor-man's" multi-segmented apps and only works if all code segments in the app are built with the same set of globals.

`-prc <infile>`	Input existing .prc file. Usually used in combination with the -patch option (to patch object code) or the `-oRcp` option (to dump out existing *.prc* resources in *.rcp* text format).
`-l <language>`	Compile resources for a specific language.
`-I <path>`	Search path for bitmap and include files. More than one -I <path> option may be given. The current directory is always searched.
`-F <encoding>`	Compile resources for a specific font encoding. <encoding> can be: Jp (Shift-JIS), B5 (Big-5), Hb (Hebrew), Ge, Sp, It, or Fr.
`-maxBmpDepth <depth>`	Don't include any bitmaps greater than the given depth (1, 2, 4, 8, 16, 24, 32 allowed).

Modification options:

`-patch <resType> <resID> <patchFile>`	Patch given resource using info from <patchFile> which is a text file with this format:
	
	Version 1.0:
	[; comment]
	<offset> <oldData> <newData> [;comment]
	<offset> <oldData> <newData> [;comment]
	
	where all numbers are in hex.
	
	Example:
	`16A 001122 445566`
	`22A ABBCDE ABBCDF`
	
	--- 	
	
	`Version 2.0:`
	`#PATCH_VERSION=2.0`
	`[; comment]`
	`<offset>.L, <old1>.W <old2>.W ..., <new1>.W <new2>.W ...`
	`<offset>.L, <old1>.W <old2>.W ..., <new1>.W <new2>.W ...`
	
	where .L is a 32-bit value and .W is a 16 bit value. Simple addition, subtraction, and parentheses are also supported.

`-del <resType> <resID>`	Remove given resource from output *prc*. This option is most useful when reading in an existing *prc* to create a new one.

Output types:

`-o <outfile>`	(default: "`out.prc`") Output file name.
`-oRcp <name>`	Generate .RCP and .BMP output files from resources. Usually used in conjunction with the `-prc` option to convert resources from existing Palm OS *.prc*s into source .RCP text files.
`-overlayOf <name> <langID> <countryID>`	This option tells Palm-RC to construct the output file (specified by the `-o` option) as an overlay database for the base *prc* file `<name>`. When this option is present, an 'ovly' resource will be inserted (or the existing one updated) into the `<name>` base *prc* file and the output *prc* file will have a corresponding 'ovly' resource installed into it as well. When this option is specified, the type and creator of the output *prc* are obtained from the base *prc*, so the `-type` and `-cr` options must not also be present. The 'ovly' resource in the base *prc* basically contains a list of resources in the base *prc* along with their lengths and checksums. The 'ovly' resource in the output *prc* file contains a list of resource in the overlay database. When the database is opened, the PalmOS uses the information in the base and overlay database overlay resources to validate that the overlay is in fact a valid overlay of the base *prc*.

Output options:

`-name <prcName>`	(default: "`out`") PalmOS name for *prc* file.
`-cr <creator>`	(default: `????`) 4 character creator for *prc* file.
`-type <type>`	(default: `appl`) 4 character type for *prc* file.
`-version <version>`	(default: `1`) version number of *prc* file.
`-zCrDate`	Set creation date to 0 (for testing).
`-zModDate`	Set modification date to 0 (for testing).
`-incBaseOverlay`	Include a base overlay resource on by default except with `-overlayOf` option.
`-incBaseOverlay-`	Don't include a base overlay resource.

prc flags:

`-backup`	Set backup bit in *prc* file.
`-hidden`	Set hidden bit in *prc* file.
`-readOnly`	Set readOnly bit in *prc* file.
`-resetAfterInstall`	Set resetAfterInstall bit in *prc* file.
`-copyPrevention`	Set copyPrevention bit in *prc* file.

Diagnostic options:

`-v`	Verbose mode.
`-ignoreDups`	Don't print warnings about duplicate resources.

Notes:

Resource Object *(.ro)* Files

When compiling resources, there are two options for output: a normal *.prc* file or "resource object" file. A *.prc* file is complete and ready to be run on a device. A resource object file is an "incomplete prc," a resource database which consists only of the resources defined in *.rcp* and *.rsrc* files. Resource object files are marked by the extension *".ro*" and have creator 'PlRC' and type 'reso'.

A resource object file is taken as input by the build-prc tool. If you want to take advantage of the multi-segment support or other advantages of build-prc that aren't included in Palm-RC, you can use Palm-RC to compile the resources and then use build-prc to add in the code resources and create the *.prc* file.

Palm-RC will automatically create a *.ro* file when it detects that there are no code resources being added.

Examples:

Creating *Myapp.prc* from linker output and resources:

```
Palm-RC -rcp ../Rsc/MyApp.rcp
        -gccApp ../Obj/MyApp.code.1.sym
        -I ../Src -I ../Rsc  -o ../Obj/MyApp.prc
        -name MyApp -cr MyAp
```

Creating *Myapp.ro* from a resource file:

```
      Palm-RC -rcp ../Rsc/MyApp.rcp  -o ../Obj/MyApp.ro
Creating a resource file from an existing .PRC:
      Palm-RC -prc ../Obc/MyApp.prc
              -oRcp MyApp.rcp
```

Patching an existing database:

```
Palm-RC -prc ../Obc/MyApp.prc
        -patch code 0001 ../Src/Patches.txt
        -o ../Obj/MyAppPatched.prc
        -name MyApp -cr MyAp
```

7.4 RCP file format

Syntax:

Items in ALL CAPS appear as literals in the file.
Items enclosed in < and > are required fields.
Items enclosed in [and] are optional fields.

Each field's required type is indicated by a suffix after the field name (see below for types).

Types:

```
.i  = identifier
      example:  kFoo
.s or .ss  = string (may contain the following character escapes: \t (tab) \n (cr)
    or \nnn where nnn is an octal character code, or \xXX where XX is a
    hex character code)
      example:  "Click Me\t\015\x0A"
    may be a multi line string.  Palm-RC will concatenate strings on seperate
lines
      enclosed with quotes and terminated by the \ character
      example:  "Now is the time for all good "\
                "men to come to the aid of their country"
    no difference between .s and .ss (used to differentiate single-line from
multiline)

.n = number, defined constant or simple arithmetic expression.  Valid operators
are + - * /.
      precedence is left to right. math is integer.
      examples: 23 12+3+1 12*4 14*3+5/2

.p = position coordinate.  Which may be either a number, expression or one
     of the following keywords
        AUTO       : Automatic width or height.  The width/height of the
                     item is computed based on the text in the item.
                    valid only for widths or heights of items.
        CENTER     : Centers the item either horizontally or vertically.
                     Only valid for left or top coordinate of an item.
       PREVLEFT   : Previous items left coordinate
       PREVRIGHT  : Previous items right coordinate
       PREVWIDTH  : Previous items width
       PREVTOP    : Previous items top coordinate
       PREVBOTTOM : Previous items bottom coordinate
       PREVHEIGHT : Previous items height

       Example: PREVRIGHT+2

       NOTE: AUTO and CENTER must stand alone and are not valid in arithmetic expressions
```

Comments:

Comments are single line and begin with //.

Note that they are only allowed at the outer scope. Comments within a FORM or MENU or other command are treated as errors.

Quoting:

Strings enclosed in single quotes are quoted exactly--backslashes are treated as backslashes, not as quoting characters.

Example: "`hello\tthere`" contains a tab, but `'hello\tthere'` doesn't.

7.4.1 Include Files

The *.rcp* file may contain `#include` directives. This allows a programmer to have one header file for the project containing resource IDs. Source code can reference the symbols as can Palm-RC.

Palm-RC understands two include file formats. Following is the fairly limited syntax for each of them:

`.h`	`#define <Symbol.i> <Value.n>`
`.inc`	`<Symbol.i> equ <Value.n>`

Each symbol may then be used in place of any number.
Note: `#ifdefs` are ignored by Palm-RC.

7.5 Resource Language Reference

The *.rcp* file may contain the following commands:

`FORM`	PalmOS Form
`MENU`	Menu bar
`ALERT`	Alert dialog box
`VERSION`	Version string
`STRING`	String
`STRINGTABLE`	String table
`APPINFOSTRINGS`	Category name strings
`APPLICATIONICONNAME`	Application Icon's Name
`APPDEFAULTCATEGORY`	Application Default Category
`APPLICATION`	Application four character type
`ICON`	Icon bitmap
`BITMAP`	Bitmap
`BOOTBITMAP`	Boot Bitmap
`TRANSLATION`	Language string translation
`DATA`	Include raw data from a file as a resource
`HEX`	Defined resource as hex data

FORM (tFRM)

```
FORM ID <FormResourceId.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>)
[FRAME]
[NOFRAME]
[MODAL]
[SAVEBEHIND]
[USABLE]
[HELPID <HelpId.n>]
[DEFAULTBTNID <BtnId.n>]
[MENUID <MenuId.n>]
BEGIN
      <OBJECTS>
END
```

<OBJECTS>: one or more of:

TITLE	<Title.s>
BUTTON	{<Label.s> | GRAPHIC <Id.n>} ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FRAME] [NOFRAME] [BOLDFRAME] [FONT <FontId.n>]
PUSHBUTTON	{<Label.s> | GRAPHIC <Id.n>} ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FONT <FontId>] [GROUP <GroupId.n>]
CHECKBOX	<Label.s> ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FONT <FontId>] [GROUP <GroupId.n>] [CHECKED]
POPUPTRIGGER	<Label.s> ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FONT <FontId.n>]
SELECTORTRIGGER	<Label.s> ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FONT <FontId.n>]
REPEATBUTTON	<Label.s> ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR] [RIGHTANCHOR] [FRAME] [NOFRAME] [BOLDFRAME] [FONT <FontId.n>]
SCROLLBAR	ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [VALUE <Value.n>] [MINVALUE <Min.n>] [MAXVALUE <Max.n>] [PAGESIZE <PageSize.n>]
LABEL	<Label.s> ID <Id.n> AT (<Left.p> <Top.p>)[USABLE] [NONUSABLE] [FONT <FontId.n>]
FIELD	ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>)[USABLE] [NONUSABLE] [DISABLED] [LEFTALIGN] [RIGHTALIGN] [FONT <FontId.n>] [EDITABLE] [NONEDITABLE] [UNDERLINED] [SINGLELINE] [MULTIPLELINES] [MAXCHARS <MaxChars.n>] [AUTOSHIFT] [HASSCROLLBAR] [NUMERIC]
POPUPLIST	ID <Id.n> <IdList.n>

LIST	<Item.s> <Item2.s>... ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABLE] [DISABLED] [VISIBLEITEMS <NumVisItems.n>] [FONT <FontId.n>]
FORMBITMAP	AT (<Left.p> <Top.p>) BITMAP <BitmapId.n> [NONUSABLE]
GADGET	ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) [USABLE] [NONUSABE]
TABLE	ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>) ROWS <NumRows.n> COLUMNS <NumCols.n> COLUMNWIDTHS <Col1Width.n> <Col2Width.n.>...
GRAFFITISTATEINDICATOR	AT (<Left.p> <Top.p>)
Notes:	The bitmap referenced by FORMBITMAP must appear as a seperate resource in the rcp file via the BITMAP command. MAXCHARS is required for FIELD to work properly.

Example:

```
FORM ID 1 AT (2 2 156 156)
USABLE
MODAL
HELPID 1
MENUID 1
BEGIN
  TITLE "AlarmHack"
  LABEL "Repeat Datebook alarm sound" ID 2000) AT (CENTER 16)
  PUSHBUTTON "1" ID 2001 AT (20 PrevBottom+2 12) AUTO GROUP 1
  PUSHBUTTON "2" ID 2002 AT (PrevRight+1  PrevTop PrevWidth PrevHeight) GROUP 1
  PUSHBUTTON "3" ID 2003 AT (PrevRight+1  PrevTop PrevWidth PrevHeight) GROUP 1
  PUSHBUTTON GRAPHIC 1000 ID 2004 AT (PrevRight+1  PrevTop PrevWidth PrevHeight)
GROUP 1

  LABEL "times.  Ring again every" ID 601 AT(CENTER PrevBottom+2) FONT 0

  PUSHBUTTON "never" ID 3000 AT (13 PrevBottom+2 32 12) GROUP 2
  PUSHBUTTON "10 sec" ID 3001 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP 2
  PUSHBUTTON "30 sec" ID 3002 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP 2
  PUSHBUTTON "1 min" ID 3003 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP 2

  LABEL "Alarm sound:" ID 601 AT (24 PrevBottom+4)
  POPUPTRIGGER "          " ID 5000 AT (PrevRight+4 PrevTop 62 AUTO) LEFTANCHOR
  LIST "Standard" "Skip Along" "Beethoven" "EuroCop" "Cricket" "Bleep" "Computer2"
ID 6000 AT (PrevLeft PrevTop 52 1) VISIBLEITEMS 10 NONUSABLE
  POPUPLIST ID 5000 6000

  BUTTON "Test" ID 1202 AT (CENTER 138 AUTO AUTO)
  GRAFFITISTATEINDICATOR  AT (100 100)
END
BITMAP ID 1000        "PushButton.1.bmp"
```

MENU (MBAR)

```
MENU ID <MenuResourceId>
BEGIN
	<PULLDOWNS>
END

<PULLDOWNS>: one or more of:

PULLDOWN <PulldownTitle.s>
BEGIN
	<MENUITEMS>
END

<MENUITEMS>: one or more of:
	MENUITEM [HIDDEN] <MenuItem.s> <MenuItemId.n> [AccelChar.c]
```

Example:

```
MENU ID 100
BEGIN
	PULLDOWN "File"
	BEGIN
		MENUITEM "Open..." ID 100 "O"
		MENUITEM "Close" ID 101 "C"
	END
	PULLDOWN "Options"
	BEGIN
		MENUITEM "Get Info..." ID 500 "I"
	END
END
```

ALERT (tALT)

```
ALERT ID <AlertResrouceId.n>
[HELPID <HelpId.n>]
[INFORMATION] [CONFIRMATION] [WARNING] [ERROR]
BEGIN
	TITLE <Title.s>
	MESSAGE <Message.ss>
	BUTTONS <Button.s> <BUTTON.s>...
END
```

Example:

```
ALERT ID 1000
HELPID 100
CONFIRMATION
BEGIN
	TITLE "AlarmHack"
	MESSAGE "Continuing will cause you 7 years of bad luck\n"\
		"Are you sure?"
	BUTTONS "Ok" "Cancel"
END
```

VERSION (tver)

```
VERSION ID <VersionResourceId.n> <Version.s>
```

Example:

```
VERSION ID 1 "0.09"
```

STRING (tSTR)

```
STRING ID <StringResourceId.n> <String.ss>
```

Example:

```
STRING ID 100 "This is a very long string that demonstrates carriage returns\n" \
       "as well as continued .ss syntax strings"
```

STRINGTABLE (tSTL)

```
STRINGTABLE ID <StringTableResourceId.n>  <PrefixString.ss> <String.ss> ...
<String.ss>
```

`STRINGTABLE` is intended for null-terminated strings. The terminators do not need to be explicitly added (see Examples below). If null characters are embedded into a string, only the portion of the string up to the first null character will be placed in the resource; a warning will also be issued.

To omit the `PrefixString`, start the `STRINGTABLE` with an empty set of quotes.

Example (with prefix):

```
STRINGTABLE ID 1000 "Directions" "North" "South" "East" "West"
```

Example (without prefix):

```
STRINGTABLE ID 1000 "" "foo" "bar" "baz"
```

Example (without prefix, vertical style):

```
STRINGTABLE ID 1000 ""
  "foo"
  "bar"
  "baz"
  "con" \
  "catenated"
```

APPINFOSTRINGS (tAIS)

The first 16 of these are used for localized category names. After that, it's up to the application developer. Currently, palm-rc enforces that there must be at least 16 of these (because the system requires it for categories)

```
APPINFOSTRINGS ID <AppInfoStringResourceId.n>
       <Category.s>
       <Category.s>
       ...
```

Example:

```
APPINFOSTRINGS ID 1000
        "Unfiled"
        "System"
        "Games"
        "Utilities"
        "Main"
        ""
        ""
        ""
        ""
        ""
        ""
        ""
        ""
        ""
        ""
        ""
```

APPLICATIONICONNAME (tAIN)

```
APPLICATIONICONNAME ID <AINResourceId.n> <ApplicationName.s>
```

Example:

```
APPLICATIONICONNAME ID 100 "AlarmHack"
```

APPDEFAULTCATEGORY (taic)

```
APPDEFAULTCATEGORY <ApplicationCategory.s>
```

On OS 3.5 and up, this will let the app developer specify which category the application should show up in when installed. This should be used with care, as the user expects that applications show up in "Unfiled". **If you are specifying one of the built-in categories, you should specify the category in English EVEN IN LOCALIZED VERSIONS**. The launcher knows how to do the translation.

On pre-Palm OS 3.5 systems, the resource generated by this command is ignored.

Example:

```
APPDEFAULTCATEGORY "Games"
```

APPLICATION (APPL)

```
APPLICATION ID <ApplResourceId.n> <APPL.s>
<APPL.s> must be 4 characters long
```

Example:

```
APPLICATION ID 1 "ALHK"
```

ICON (tAIB)

Converts a Microsoft Windows-style bitmap(s) to tAIB Palm OS icon resources with ID 1000. If one or more `DEPTHx` is specified, then the icon resource will include the other specified bit depths in addition to the 1-bit deep version. The source bitmap files can be in 1, 2, 4, 8, or 24 bits per pixel format and Palm-RC will do the necessary bit-depth conversion.

If the `TRANSPARENT` keyword is present, then the bitmap will be created such that any pixels matching the given color will be transparent. The transparency applies to all bitmaps in the family (except 1-bit). In schemes with a color table (2, 4, and 8 bit depth), the index that will be made transparent is the one that matches the color specified most closely. NOTE: 2-bit images will change from version 1 format to version 2 format if you specify transparency.

IMPORTANT: Bit depths greater than 2 bits per pixel are only supported in Palm OS 3.5 or greater and DEPTH16 is only supported on the Handspring platform.

```
ICON <IconFileName.s>
       [DEPTH2 <IconFileName2.s> ]
       [DEPTH4 <IconFileName4.s> ]
       [DEPTH8 <IconFileName8.s> ]
       [DEPTH16 <IconFileName16.s> ]
       [TRANSPARENT <red.n> <green.n> <blue.n>]
```

Example:

```
ICON "myicon.bmp" DEPTH2 "myicon.2.bmp" TRANSPARENT 0 255 0
```

SMICON (tAIB)

Converts a Microsoft Windows-style bitmap(s) to `tAIB` Palm OS icon resources with ID 1001. If one or more `DEPTHx` is specified, then the icon resource will include the other specified bit depths in addition to the 1-bit deep version. The source bitmap files can be in 1, 2, 4, 8, or 24 bits per pixel format and Palm-RC will do the necessary bit-depth conversion.

If the `TRANSPARENT` keyword is present, then the bitmap will be created such that any pixels matching the given color will be transparent. The transparency applies to all bitmaps in the family (except 1-bit). In schemes with a color table (2, 4, and 8 bit depth), the index that will be made transparent is the one that matches the color specified most closely. **Note:** 2-bit images will change from version 1 format to version 2 format if you specify transparency.

IMPORTANT: Bit depths greater than 2 bits per pixel are only supported in Palm OS 3.5 or greater and DEPTH16 is only supported on the Handspring platform.

```
SMICON <IconFileName.s>
       [DEPTH2 <IconFileName2.s> ]
       [DEPTH4 <IconFileName4.s> ]
       [DEPTH8 <IconFileName8.s> ]
       [DEPTH16 <IconFileName16.s> ]
       [TRANSPARENT <red.n> <green.n> <blue.n>]
```

Example:

```
SMICON "mysmicon.bmp" TRANSPARENT 0 255 0
```

BITMAP (Tbmp), BOOTBITMAP (Tbsb)

Converts Microsoft Windows-style bitmap to TBmp Palm OS bitmap resources. If one or more DEPTHx is specified, then the icon resource will include the other specified bit depths in addition to the 1-bit deep version. The source bitmap files can be in 1, 2, 4, 8, or 24 bits per pixel format and Palm-RC will do the necessary bit-depth conversion.

If the TRANSPARENT keyword is present, then the bitmap will be created such that any pixels matching the given color will be transparent. The transparency applies to all bitmaps in the family (except 1-bit). In schemes with a color table (2, 4, and 8 bit depth), the index that will be made transparent is the one that matches the color specified most closely. NOTE: 2-bit images will change from version 1 format to version 2 format if you specify transparency.

You may control the compression of bitmaps using NOCOMPRESSS, COMPRESS, or FORCECOMPRESS. NOCOMPRESS means that the bitmap won't be compressed no matter what. COMPRESS (the default) will compress the bitmap if the resulting bitmap is smaller. FORCECOMPRESS will always compress the bitmap. 1-bit and 2-bit without transparency (aka version 1 bitmaps) will use scanline compression. 2-bit with transparency and all 4-bit, 8-bit, and 16-bit bitmaps will use the smaller of scanline and RLE compression. Using the optional depth specification, you can specify which depths a command applies to (1, 2, 4, 8, 16, 24, 32). If no depth specification is present, the command will apply to all depths. Multiple depth commands can be present and will be evaluated left to right.

If the INCLUDECLUT keyword is present, then the bitmap will include it's own color lookup table copied from the original bitmap file.

IMPORTANT: Bit depths greater than 2 bits per pixel are only supported in PalmOS 3.5 or greater and DEPTH16 is only supported on the Handspring platform.

```
BITMAP ID <BitmapResourceId.n>  [INCLUDECLUT]

        [        <BitmapFileName.s>    ]
       [DEPTH2  <BitmapFileName2.s>  ]
       [DEPTH4  <BitmapFileName4.s>  ]
       [DEPTH8  <BitmapFileName8.s>  ]
       [DEPTH16 <BitmapFileName16.s> ]
       [TRANSPARENT <red.n> <green.n> <blue.n>]
       [NOCOMPRESS [<depth.n>]* | COMPRESS [<depth.n>]* | FORCECOMPRESS
[<depth.n>]*]*
```

Example:

```
BITMAP ID 100 "mybitmap.bmp" DEPTH2 "mybitmap.2.bmp" TRANSPARENT 0 255 0
FORCECOMPRESS  NOCOMPRESS 2
```

DATA

Includes an entire file as a user-defined resource

```
DATA <ResType.s> ID <ResId.n> <FileName.s>
```

Example:

```
DATA "dflt" ID 1 "DefaultDataDB.bin"
```

HEX

Define a user-defined resource as hex and/or ascii data

```
HEX <ResType.s> ID <ResId.n> <Byte.n> | <String.s>...
```

Example:

```
HEX "junk" ID 1000
       0x00 0x00 0x00 0x23 "String" 0x00 "String2"
       0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

7.6 International Support

Palm-RC supports a limited form of international tokenization. It works by substituting strings in the resource definitions with replacements specified in a `TRANSLATION` section. Multiple translation blocks may be specified in a resource script. The active language is specified with the -l flag to Palm-RC.

The biggest problem is with positioning of controls. If you use `AUTO`, `CENTER,` and `PREVRIGHT, for example,` it might not involve any position changing in your script. If you do need to change the position, right now the only workaround is to put some `#ifdefs` in your file and hook up a custom rule to preprocess your source file. Another way to do internationalizaton is to have multiple resource files compiled conditionally depending on the build target.

Note: The `TRANSLATION` section must appear first in the *.rcp* file, before any use of the strings to be translated.

Example:

```
Palm-RC -l FRENCH myscript.rcp
```

TRANSLATION

```
TRANSLATION <Language.s>
BEGIN
      <STRINGTRANSLATIONS>
END
Where <STRINGTRANSLATINOS> is one or more of:
<Original.s> = <Translated.ss>
```

Tip*:* For long strings, define a short keyword and then define both native and foreign translations for it.

Example:

```
TRANSLATION "FRENCH"
BEGIN
      "Repeat Datebook alarm sound" = "Répétitions Alarme Agenda"
      "Ring again every" = "Rappel tous les"
END
```

7.7 Known Bugs

- `LIST: DISABLED` doesn't work. Seems to be a Palm OS bug.
- `LIST: VISIBLEITEMS` may be required for list objects to show properly.
- `FIELD: MAXCHARS` required for field control to accept characters to work.
- Comments are only allowed at the outermost scope level. Comments within `FORM, MENU` or other commands are syntax errors.

8 Palm Debugger User's Guide

8.1 About PalmDebugger

PalmDebugger is a Windows and Macintosh desktop application for debugging Palm OS executables. It provides assembly- and source-level debugging as well as support for managing databases on the Palm OS device. Palm Debugger can communicate with a Palm OS device over serial or USB ports, or with the Palm OS Emulator desktop application (which acts as a virtual Palm OS device) over TCP/IP.

All Palm OS devices have a debugger stub in their ROM with which PalmDebugger communicates. This debugger stub, although fairly small, provides enough basic support for all of PalmDebugger's functions, including the display and modification of memory, setting breakpoints, single-stepping, dumping memory heaps, and modifying databases.

Besides basic debugging support of Palm OS executables, PalmDebugger also provides a Console window for system administration functions. The Console window behaves like a Unix shell or DOS command line interface to the device. Included are Console window commands for such tasks as getting a directory of databases on the device, creating and deleting databases, importing and exporting databases to and from the desktop computer, and many other system-level functions.

PalmDebugger is a developer's tool. It is very powerful but, unfortunately, is not a very "polished" application. In it's design, user-friendliness has taken a back seat to power and functionality. But time spent learning how to use it will be time well spent and it will pay for itself many times over. With the help of this document, you should be able to learn the basics of how using Palm Debugger within an hour or so.

8.2 User Interface Overview

8.2.1 The Windows

When launched, PalmDebugger displays four windows:

- *Debugger* window: The Debugger window is used for assembly-level debugging.
- *Console* window: The Console window is used as a command-line based shell for managing databases on the device.
- *CPU Registers* window: The CPU Registers window is used for assembly-level debugging.
- *Source* window: The Source window is used for source-level debugging. Note that the Source window and source-level debugging support is only present in the Windows version.

Most of the powerful functions of the debugger are accessed by typing commands either into the Debugger or Console windows. There are, however some menus for performing basic source-level debugging functions such as setting breakpoints or single-stepping. The Source window is essentially an "output-only" window that is used to display the source code and local variables for the executable currently being debugged. The CPU Registers window is also an output-only window that displays the values of each of the CPU registers while debugging.

Commands are entered into the Debugger and Console windows by typing the command, then hitting the [Enter] key (Note: on the Macintosh, use the [Enter] key on the numeric keyboard or [Cmd-Return]*).* Both the Debugger and Console windows have a 'help' command that displays a list of possible commands. Help on any specific command can be displayed by entering '`help` *`cmdName`*.'

The Debugger and Console windows behave like edit windows. They support cut, copy, paste, and undo (undo is available in the Windows version only). Whenever you hit the *Enter* key, the window executes whatever text is currently selected, or the entire current line if the selection is empty. If more than one line is selected, every selected line will be executed. If you simply want to create a new line without executing the current line, hit the [Return] (on a Macintosh), or [Shift-Enter] (on Windows).

The Windows version also supports the following special key sequences: [Shift-Backspace] will delete all text from the current selection point to the end, and the undo command [Ctrl-Z] entered immediately after executing a command will delete the output of the command from the window.

8.2.2 The Menus

Most of the menu items in the **File** menu are not yet implemented. In the future, this menu will allow saving and printing of window contents.

The **Edit** menu provides the usual *cut*, *copy*, *paste* functions as well as *undo* and *redo* and a *font* command for changing the font used in all of the windows.

The **Connection** menu is used to set up communications with the actual or virtual (in the case of PalmOSEmulator) device. The choices are through:

- One of the serial ports (COM1 through COM4 on Windows, Modem or Printer on Macintosh)
- USB (Windows 98 and Windows 2000)
- Emulator (to communicate with the PalmOSEmulator application running on the desktop)

When one of the serial ports is selected, the menu items for changing the baud rate are enabled.

The **Source** menu provides basic source-level debugging commands, such as breakpoints, single stepping, or continuing, as well as commands for setting up the symbol files for source-level debugging.

The **Window** menu (Windows version only) provides the standard Windows commands for selecting various windows and rearranging them.

8.3 The Console Window

The Console window behaves like a Unix shell or DOS command line interface to the device. Through the Console window, you get a directory of databases on the device, create and delete databases, transfer databases to and from the desktop, and display system information.

In order to use the Console window to communicate with the device, the device must be running a "console stub". The console stub runs as a background thread on the device and waits for commands over the serial or USB port, processes the commands, then sends back responses. Because the console stub runs as a background thread, it does not affect the normal operation of the device, and applications can be used normally while the thread is running. However, because it requires memory and system resources, it is not normally started unless specifically activated by the user. Once started, the console stub continues to run, and prevents other applications (like HotSync) from using the serial port until the device is soft-reset.

To activate the console stub on a device, enter the Graffiti sequence *Shortcut-.-2,* i.e., a shortcut stroke (a script lower case letter 'L'), followed by a period (two single taps), followed by the number 2 (entered in the right side of the Graffiti area). When the console stub first starts, it sends out a "Ready..." message. If you have PalmDebugger running and connected with a cable to the device, this message will show up in the Console window of PalmDebugger. If the console stub was already started or if PalmDebugger was not connected to the device, you will not see the message. Note that if you are debugging using the PalmOSEmulator instead of a real device, you do not need to manually start the console stub because it will be started for you automatically.

Be sure to double-check the communications method used in the Connection menu. If debugging with a Handspring device, the default method should be set to "USB". All other Palm devices use serial at 57,600 baud. If desired, you can debug a Handspring device using serial instead of USB by pressing the "up" key on the device when starting the console stub (hold down the [Up] key as you type the Graffiti shortcut sequence).

Once the console stub is running on the device, you should be able to successfully enter commands in the Console window of PalmDebugger and see results back from the device. A simple, quick command to try is "hl 0". This displays a list of memory heaps on card #0 of the device. Here's an example output from the `hl` command:

```
hl 0
   index   heapID  heapPtr     size     free  maxFree    flags
-------------------------------------------------------------
       0     0000 00001480 00016B80 00010A90 0001046C     8000
       1     0001 1001810E 001E7EF2 001E53C0 001E5342     8000
       2     0002 10C08212 00118DEE 0000A01C 0000A014     8001
```

If the console stub is not running on the device, or if the serial or USB connection is not correct, you will see the following error message after a few seconds:

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

If you see this message, double-check that you have the correct communication options set in the Connection menu of PalmDebugger. If you are using a serial connection, make sure PalmDebugger is set to 57,600 baud, check that you have the correct *handshaking* mode, check your cable connection, and make sure that the device is powered on, and that you have started the console stub as described above.

8.3.1 Commonly used Console Commands

import

By far, the most commonly-used console command is the `import` command. This command copies a Palm OS database from the desktop to the device. It is used whenever you have built a new version of an application and want to load the application onto the device for testing. It basically performs the same function as the Installer tool provided with the HotSync application, but is much more convenient to use than the Installer tool when debugging.

The basic form of the `import` command is:

```
import <cardNo> <filename>
```

Where <*filename*> is the name of a file on the desktop. By default, PalmDebugger looks in the *Device* sub-directory within the PalmDebugger application directory for the named file. The filename can also be specified using a relative or absolute path if it's not in the Device directory. The <`cardNo`> is nearly always 0, which means import the database into the built-in RAM on the device.

Here's an example of the import command and its output:

```
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!!
```

Note that after the database is stored on the device, its name is not the same as the name of the file on the desktop. The Palm OS database name is stored within the file itself, was "*Text to Hex*" in the example shown above.

If the file you're trying to import already exists on the device, it will be deleted and replaced by the new file unless the current database is open on the device. If it is open when you try to import a new copy, you will get a `dmErrAlreadyExists (0x0219)` Data Manager error code back from the import command. For example:

```
import 0 Tex2HexApp.prc

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

### Error $00000219 occurred
```

Unfortunately, most of the error messages you see in PalmDebugger are fairly cryptic hex codes such as the one above. To determine the meaning of the error message, you will have to look up the name of the error code from the Palm OS header files. All Palm OS error codes are 16-bit values with the upper byte representing the manager and the lower byte representing a manager-specific error code. In the example above, the manager code was 0x02 and the specific error code was 0x19. The <*SystemMgr.h*> header file contains the manager codes (`0x02 is dmErrorClass`) and the manager-specific error code can be found in that manager's header file (`0x19, decimal 25, is dmErrAlreadyExists in <DataMgr.h>`).

export

The *export* command does the opposite of the import command above. It copies a database from the device to the desktop.

The basic form of the export command is:

```
export <cardNo> <filename>
```

Where <*filename*> is the name of the Palm OS database. To get a list of databases on the device to determine their names, use the *dir* command described below. The database will be copied to the desktop in the standard *.prc/.pdb* file format (these two formats are actually identical; *prc* is normally used to indicate resource databases and *.pdb* is normally used to represent record databases) and will be given a name of <*filename*> without any added extension. All exported files are placed into the Device sub-directory of PalmDebugger.

Here's an example of the export command and its output. Note that you must use quotes if there are spaces in the name:

```
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!!
```

The above command will create a file called *Text to Hex* within the Device sub-directory of PalmDebugger.

dir

The *dir* command will display a list of databases on the device. The basic form of this command is:

```
dir <cardNo>|<searchOptions> [<displayOptions>...]
```

The <*displayOptions*> are usually left blank, or the `-a` option is specified, which means to display all information. For a complete list of options, type `'help dir'` in the Console window.

Here's an example of the command and its (abbreviated) output:

```
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
```

del

The *del* command will delete a database from the device. The basic form of this command is:

```
del <cardNo> <filename>
```

Where *<filename>* is the name of the Palm OS database. To get a list of databases on the device to determine their names, use the `dir` command described above. Keep in mind that you won't be able to delete a database if it is currently open. If you get an error from this command, such as "`##ERROR Deleting database`", the most likely reason is that the database is currently open. If this is an application that is currently running, switch to a different application and try again.

8.3.2 Less Commonly-used Console Commands

Entering `help` in the Console window will display a relatively large list of available console commands. The commands mentioned above will be used most of the time. The remaining console commands are used much less often, and most users will probably never use most of them.

Rather than describe all of the console commands in detail, we will just provide a brief overview of those most likely to be used. Later in this document, we describe how to track down and solve typical application problems, and we will describe specific features of the console commands that are useful for tracking down specific problems.

Of the remaining console commands, those dealing with memory heaps are probably used most frequently. These include the following:

`hl <cardNo>`	Display list of memory heaps
`hd <hex heapID>`	Dump a specific heap

The `hl` command displays a list of memory heaps. For example:

```
hl 0
   index   heapID  heapPtr     size     free  maxFree    flags
-------------------------------------------------------------
       0     0000 00001480 00016B80 00010A90 0001046C     8000
       1     0001 1001810E 001E7EF2 001E53C0 001E5342     8000
       2     0002 10C08212 00118DEE 0000A01C 0000A014     8001
```

The `hd` command does a dump of a specific heap. The `<heapID>` argument can be determined by looking at the "`heapID`" column of the *h*' command. Heap number 0 is always the dynamic heap and higher number heaps represent storage RAM or ROM. 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 00001490 F0001491 00001E 000026  #0  #0    fM Alarm Table
-0000155A 00001494 F0001495 000456 00045E  #0  #0    fM Graffiti Private
-000019B8 00001498 F0001499 000012 00001A  #0  #0    fM DataMgr Protect List (DmProtectEntryPtr*)
...
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15    fM Handle Table: 'Graffiti ShortCuts'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15    fM DmOpenInfoPtr: 'Graffiti ShortCuts'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15    fM DmOpenRef: 'Graffiti ShortCuts'
-----------------------------------------------------------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010A90 bytes)
  Movable Chunks:     #52     (006040 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

The `kinfo` command displays a list of all system kernel information such as tasks, semaphores, and event groups.

kinfo <options> -all : get all info -task <id>|all : get task info -sem <id>|all : get semaphore info -tmr <id>|all : get timer info	Display system kernel info

For example:

```
kinfo -all

Task Information:
  taskID     tag  priority  stackPtr   status
  ------------------------------------------------------------------
  0001772C    AMX  #   0    00017598   Idle: Waiting for Trigger
  00017900   psys  #  30    000130B4   Running

Semaphore Information:
  semID      tag   type      initValue  curValue        nesting    ownerID
  ------------------------------------------------------------------------
  00017830   MemM  resource   #-1        #1  (free)      #0        00000000
  00017864   SlkM  counting   #1         #1  (avail.)    #0        00000000
  000178CC   SndM  counting   #1         #1  (avail.)    #0        00000000
  00017968   Sync  resource   #-1        #1  (free)      #0        00000000
  00017A38   SerM  counting   #0         #0  (unavail.)  #0        00000000

Timer Information:
  tmrID      tag   ticksLeft   period   procPtr
  ---------------------------------------------
  000177FC   psys    # 493    #   0    10C6C618
```

Finally, the `mdebug` command puts the device into various modes for helping to track down memory corruption problems. Note that turning these options on slows down the device, (often considerably):

mdebug [options..] Shortcuts: -full : Full checking (slowest) -partial : Partial checking (faster) -off : No checking (fastest) Fine Tuning: Which heaps are checked/scrambled: -a : check/scramble ALL heaps each time -a- : check/scramble affected heap only Heap Checking: -c : check heap(s) on some Mem calls -ca : check heap(s) on every Mem call -c- : turn off heap checking Heap Scrambling: -s : scramble heap(s) on some Mem calls -sa : scramble heap(s) every Mem call -s- : turn off heap scramble Free Chunk Checking: -f : check free chunk contents -f- : don't check free chunk contents Min Dynamic Heap free space recording: (Recorded in the global GMemMinDynHeapFree) -min : record minimum free space in Dynamic heap -min- : don't record minimum free space	Turn on/off various memory debugging options

8.4 The Debugger Window

The Debugger window is used in conjunction with the CPU Registers window for assembly-level debugging. Commands are entered into this window for displaying and modifying memory, single stepping, setting and clearing breakpoints, dumping memory heaps, displaying database directories, automatically breaking on one or more Palm OS system calls, breaking when a memory location is modified, etc. Besides applications, this window can also debug system code, extensions, shared libraries, background threads and interrupt handlers.

The Debugger window also supports custom-defined command aliases, script files, and data structure templates, as described in more detail in the Utility Commands section below. You can automatically load these custom definitions every time you launch PalmDebugger by adding them to the *UserStartup-PalmDebugger* text file, which is executed by PalmDebugger every time it starts up.

8.4.1 Attaching to the Device

Most Debugger window commands (except for help and some utility commands) only execute when the device is connected to the desktop and halted in its debugger stub. When the device is halted in the debugger stub, it will not respond to pen taps on the screen or key presses and you will see a tiny flashing square about 4 pixels across in the upper left corner of the display (Note: Color device screens will invert and there will not be any flashing cursor). There are two main ways to put the device into this mode (besides encountering a bug of course!):

1. Enter the `Shortcut-.-1` sequence using Graffiti.
2. Compile a `DbgBreak()` call into your application and run the application until you encounter the `DbgBreak()` call.

To use method #1 above, enter the Graffiti sequence `Shortcut-1`, *i.e.*, a shortcut stroke (a script lower case letter 'L'), followed by a period (two single taps), followed by the number 1 (entered in the right side of the Graffiti area).

To use method #2 above, you must have previously entered the debugger at least once using method #1. If the debugger has not been entered at least once already using method #1, then, instead of entering the debugger, the `DbgBreak()` call will display a fatal error dialog. Alternately, you can set the low memory global `GDbgWasEntered` to non-zero before the `DbgBreak()` call. This will effectively make the device "think" it has entered the debugger already. For example:

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

If the PalmDebugger application is running and connected with the appropriate cable to the device when it halts into the debugger, you will see a message similar to the following appear in the Debugger window:

```
EXCEPTION ID = $A0
'PrvHandleEvent'
+$062C 10C0F2AA *MOVEQ.L #$01,D0        | 7001
```

Be sure to set the communications method correctly in the Connection menu of PalmDebugger. If debugging with the PalmOSEmulator, it should be set to Emulator. If debugging a Handspring device, it should be set to USB. All other Palm devices use serial connections at 57,600 baud. If desired, you can debug a Handspring device using serial by pressing the [Up] key when entering the debugger for the first time (i.e., hold down the [Up] key while entering the Graffiti shortcut sequence). If using serial, check that you have a serial cable with handshaking lines. If you don't, turn off the Handshaking option in the Connection menu.

If the PalmDebugger was not running or connected correctly to the device when it halted, you can use the `att` command to attach PalmDebugger to the device. For example:

```
att
EXCEPTION ID = $A0
+$062C 10C0F2AA *MOVEQ.L #$01,D0         | 7001
```

The `att` command sends a request packet to the device and waits for a response from the device saying where it is currently halted. If no response is received within a couple of seconds, the `att` command will timeout and display a timeout error message. This means either that the device is not halted in the debugger or that there is a problem with the connection between desktop and the device.

If you try to execute a debugger command when the PalmDebugger thinks it is not attached to the device, it will display an error message "`Error: not attached to remote`". If this happens, ensure that the device is halted in the debugger, then issue the `att` command to re-attach PalmDebugger to the device.

8.4.2 Commonly-Used Debugger Commands

This section illustrates the most commonly-used debugger commands. It does not cover the entire set of available commands or options, however. To see the entire set of available commands enter the `help` command in the Debugger window. To get help on a particular command, enter `help cmdName`.

8.4.2.1 Entering Commands

As mentioned previously, debugger commands are typed into the Debugger window and executed by hitting the *Enter* key. For commands that take arguments, such as an address on which to operate, the arguments can be entered in either decimal or hex or as expressions. In addition, expressions can be written in terms of processor registers. By default, all numbers are assumed to be in hex unless preceded by a '**#**' sign (decimal) or '%' sign (binary).

A number of commands allow you to intelligently repeat the command simply by pressing [Enter] repeatedly. The 'dm' command is one of these commands - every time you hit [Enter] it displays the next 16 bytes of data. The 's' and 't' commands for single stepping also repeat when you hit Enter.

One shortcut character is introduced below: the '.' character. This is a shorthand representation for the address value used for the last command. For a full description of expression evaluation and all the shorthand characters that are available, see the Debugger Expressions section below.

The following is an example that shows how to display and disassemble memory using various expressions. In all the examples in this section, commands entered by the user are shown in **bold** and comments are show in *italics*:

```
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     <=Shows how to enter an expression for an address
00000120: 6F BC 00 00 07 22 00 00 00 06 00 01 7D 72 00 FD "o....".....}r.."

dm .+10       <=Illustrates using the '.' character to represent
                the last address entered.
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
```

You can even enter an expression by itself on the command line as a form of hex calculator:

```
20*4+3
$83 #131 #-125 '.'
```

This has been a brief introduction to expressions in order to lead into the commonly-used commands. For a full discussion of expressions and the available operators, see the Debugger Expressions section below.

8.4.2.2 Displaying Registers, Memory and Instructions

Usually, the first thing one does after breaking into the debugger is to disassemble code around the current program counter, or display variables. The following commands perform these functions, as well as allowing you to change memory.

One fairly advanced feature of the debugger is the ability to dump memory according to a specific structure definition. This is the <template> option to the dm command. The section below on Utility Commands describes how to define templates for various structures.

dm <address> [<count>] [<template>]	Display memory at <address> for <count> bytes. If count is omitted then assume a count of 16. If <template> is included, then display memory according to that template.
db <address>	Display the byte (8 bits) at given address
dw <address>	Display the word (16 bits) at given address
dl <address>	Display the long word (32 bits) at given address
sb <address> <value...>	Set Byte(s) at given address to <value>. Entering multiple <values> will set multiple bytes starting at <address>
sw <address> <value...>	Set Word(s) (16 bits) at given address to <value>. Entering multiple <values> will set multiple words starting at <address>

`sl <address> <value...>`	Set Long(s) (32 bits) at given address to <value>. Entering multiple <values> will set multiple longs starting at <address>
`il <address> [<lineCount>]`	Instruction List. Disassemble the instructions at the given address.
`reg`	Display current values of all the processor's registers. Note that the current values of all the registers also appear in the CPU Registers window.
`sc`	Stack Crawl. Display a list of routines on the stack using information stored in the frame pointer register (A6).
`sc7`	Stack Crawl using the stack pointer (A7) instead of the frame pointer (A6). This will display information about routines on the stack that don't set up frame pointers but will also sometimes show bogus routines as well.

The following example shows how some of these commands can be used:

```
reg            <= Display all processor registers
D0 = 00000102 A0 = 10C0EEF6 USP = 420024FD
D1 = 00000013 A1 = 10C0EF0E SSP = 00013412
D2 = 0000001A A2 = 000134F0
D3 = 00000000 A3 = 00015404
D4 = 0001008E A4 = 10CD884E
D5 = 00000000 A5 = 00014A06
D6 = 00D1F334 A6 = 000134DA PC = 10C0EEFE
D7 = 0001515E A7 = 00013412 SR = tSxnzvc Int = 0

10C0EEFE *MOVEQ.L #$01,D0 | 7001

sc                    <= Display stack crawl. The routines are listed
                      in order from oldest (at top) to newest (at bottom).
                      In this example, the current routine 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

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

il pc          <= Disassemble code at 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

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>@........"
```

```
sb 0 1 2       <= Set the two bytes at address 0 to 1 & 2
Memory set starting at 00000000

dm 0 20        <= Display 0x20 bytes at address 0
00000000: 01 02 FF FF 1A 34 3E 40 10 C0 92 D4 10 C0 92 F2 ".....4>@........"
00000010: 10 C0 93 10 10 C0 93 18 10 C0 93 20 10 C0 93 28 "........... ...("

dw 0           <= Display the word at address 0
Word at 00000000 = $0102 #258 #258 '..'

dm 0 RectangleType     <= Display memory at address 0 as a RectangleType
                       structure
00000000  struct    RectangleType
            {
00000000    PointType topLeft
              {
00000000      SWord     x         = $0102
00000002      SWord     y         = $-1
              }
00000004    PointType extent
              {
00000004      SWord     x         = $1A34
00000006      SWord     y         = $3E40
              }
            }
```

8.4.2.3 Flow Control

The most commonly used debugger commands are probably those for flow control. These include single-stepping, breakpoints, and continuing. These are summarized below along with their most commonly used options.

`g`	Go.
`s`	Step Into. Will step into subroutine calls and system traps.
`t`	Step Over. Will step over subroutine calls and system traps
`gt <address>`	Go Till address. Sets a temporary breakpoint at <address> and continues execution.
`br <address>`	Set breakpoint at <address>. PalmDebugger currently supports up to 5 breakpoints at any one time. Also note that although breakpoints are supported on ROM addresses, these types of breakpoints will force the device to run very slowly - it will essentially single step until the program counter reaches a set breakpoint address. Breakpoints set in RAM do not incur any performance penalty.
`cl [<address>]`	Clear breakpoint at <address> or, if no address, clear all breakpoints.
`brd`	Display all breakpoints.
`atb <"systemTrapName">`	A-Trap break. Set a breakpoint at the given system call.
`atc <"systemTrapName">`	A-Trap clear. Clear breakpoint at the given system call.
`atd`	Display all current A-Trap breaks.
`ss <address>`	Step Spy. Execute instructions until the DWord at <address> changes value. Warning: this routine essentially makes the processor single step through instructions until the data at <address> changes and thus makes the device run very slow.
`reset`	Perform a soft reset of the device.

The following example shows how to use most of these commands. You can follow along by first putting the device into the debugger stub by typing `Shortcut-.-1`. The commands entered by the user into the Debugger window of PalmDebugger are shown in **bold**. One of the shortcuts illustrated below is using the ':' character to represent the starting address of the current routine when entering an address. This is a very handy shortcut to use when setting breakpoints or disassembling code.

```
att            <= Attach to device
EXCEPTION ID = $A0
+$0512 10C0EEFE *MOVEQ.L #$01,D0                              | 7001

il pc          <= Disassemble 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

sc                    <= Display stack crawl. The routines are listed
                      in order from oldest (at top) to newest (at bottom).
                      In this example, the current routine 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
                      <= Just hit Enter to repeat the Single-Step
+$0694 10C0F080 *MOVEM.L (A7)+,D3-D5/A2-A4                    | 4CDF 1C38
                      <= Just hit Enter to repeat the Single-Step
+$0698 10C0F084 *UNLK A6                                      | 4E5E
                      <= Just hit Enter to repeat the Single-Step
+$069A 10C0F086 *RTS                                          | 4E75 8E53 7973 4861
                      <= Just hit Enter to repeat the Single-Step
+$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
+$0020 10CDB486 PEA -$001A(A6)                                | 486E FFE6
+$0024 10CDB48A PEA -$0018(A6)                                | 486E FFE8
+$0028 10CDB48E MOVE.L -$0024(A5),-(A7)                       | 2F2D FFDC
+$002C 10CDB492 _MenuHandleEvent ; $10C4B768                  | 4E4F A1BF
+$0030 10CDB496 TST.B D0                                      | 4A00
+$0032 10CDB498 LEA $000C(A7),A7                              | 4FEF 000C
+$0036 10CDB49C BNE.S EventLoop+$0050 ; 10CDB4B6              | 6618

gt 10cdb484           <= Go-Till address 0x10CDB484. Alternatively, we could
                      have used :+1E to represent this address
'EventLoop' Will Branch
+$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
+$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. The unit will break as soon as EvtGetEvent is
                      called.
Remote stopped due to: A-TRAP BREAK EXCEPTION
'EvtGetEvent'
+$0000 10C3B1E2 *LINK A6,$0000                             | 4E56 0000

atd           <= Display list of A-Traps
Displaying A-Traps:
Function Name Trap # Library Address
============================================================
EvtGetEvent $011D none 10C3B1E2

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

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

8.4.2.4 Heap and Database Commands

There is a set of commands that can be entered in the Debugger window that mirrors commands supported by the Console window. These include commands for displaying information about databases and memory heaps. These are especially useful because a bug often involves a corrupted memory heap or database, and you'll want to dump information about a heap or database while single-stepping through your program.

The following commands have the same options as their equivalents in the Console window and if you do a help *cmdName* on them, you will see the same help information that the Console window prints out. However, because the '-' symbol indicates subtraction in the Debugger window, it cannot be used when entering command options; instead, use the backslash symbol (\). For example:

```
dir 0 \a
```

`hl <cardNo>`	**Heap List.** Display a list of heaps for this card.
`hd <heapID>`	**Heap Dump**. Dump the contents of the given heap. The heapID can be obtained by first getting a heap list using the hl command.
`ht <heapID>`	**Heap Total.** Display just the summary information about the heap - this is the same information printed out at the end of a heap dump.
`hchk <heapID>`	**Heap Check.** Validate that the given heap is not corrupted. Both the HD and HT commands also verify a heap as part of their operation as well.
`dir <cardNo>`	**Directory**. Dump a directory of databases for the given card.

8.4.2.5 Utility Commands

The last set of commands to cover here is the utility commands. These include commands for loading memory from a file image (or vice-versa), flashing new data into the Flash ROM of a device, running debugger scripts, defining templates for use with the `dm` command, and defining debugger command aliases.

For a good example of how to define structure templates and aliases, see the SystemTypes text file which appears in the Scripts subdirectory of PalmDebugger. This text file contains definitions of a large number of Palm OS system structures.

In all cases below, the default directory for the <"`filename`"> parameter is the *Device* sub-directory of PalmDebugger except as noted.

`load <"filename"> <addr>`	Load a file into memory at the given address.
`save <"filename"> <addr> <bytes>`	Create a file in the *Device* sub-directory of PalmDebugger which is an image of the memory at the given address.
`flash <"filename"> <addr>`	Program FLASH ROM with the contents of <`filename`>.
`run <"filename">`	Execute the debugger commands contained in the text file "*filename*". The default directory is not the *Device* sub-directory like the load, save and flash commands but rather the root PalmDebugger directory.
`alias <"name"> [<"text">]`	Define or view (if no <`text`> parameter) a new alias command
`aliases`	Display list of all defined aliases.
`typedef <template> [@]<"name">`	Define a new template as the same as an existing template or as a pointer to an existing template.

`typedef struct <"name">`
`  > <template> [@]<"name">[[count]] [\-]`
`[\%]`
`typeend`
	Define a new template structure called *name*. The elements of the structure are each defined on a new line starting with '>'. The '\-' option means don't print that field. The '\%' option means print as a binary field. The [count] represents an array.

`wh [<"funcName"> | <A-trap #> | \a` `<address>]`	Where command. Find the address of a system trap by name or number or identify the memory chunk, database name, and resource type and ID that contains a particular address (the \a form).

```
load "myfile.bin" 10000              <= Load a file image into memory
100%
#24576 bytes loaded at $00010000.
save "tmpfile.bin" 10000 1000        <= Save memory contents to a file
100%
#4096 bytes saved from address $00010000 to file "tmpfile.bin"
run "SystemTypes"                    <= Execute the debugger commands
                                     contained in the text file "SystemTypes"
alias "LowMem"         "dm 0 LowMemType"
                                     <= Define a new command called "LowMem"
                                     that is an alias for the text "dm 0 LowMemType"

typedef struct "PointType"           <= Define two templates called PointType
                                     and RectangleType
       > SWord        "x"
       > SWord        "y"
       typeend

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

dm 0 RectangleType                   <= Display memory at address 0 according to
                                     the RectangleType template.
00000000  struct    RectangleType
            {
00000000    PointType topLeft
              {
00000000      SWord     x         = $-1
00000002      SWord     y         = $-1
              }
00000004    PointType extent
              {
00000004      SWord     x         = $1A34
00000006      SWord     y         = $3E40
              }
            }
wh "memhandlenew"                       <= Find the address of the given system trap
Function Name                       Trap #  Library Address
============================================================
memhandlenew                        $001E   none    10C11F4A
wh \a pc                                <= Determine the database name, resource type
                                        and resource ID that contains the current
                                        program counter.
pointer 1001B204 is 322 bytes into chunk 1001AEE2 in:
 card: 0, heapID: 1
 database: "Text to Hex", resType: 'code', resID: 1
```

8.4.3 Debugger Expressions

Any commands entered in the Debugger window can be written using expressions for one or more of the command arguments. Expressions were briefly introduced above in the command examples. This section describes the complete expression evaluation rules for reference.

All expressions must be entered without white space. White space is used to delimit parameters to commands.

The following operators are supported, and shown from highest to lowest precedence:

`.a .b .w .l .s`	unary	left to right
`~ + - @`	unary	right to left
`* /`	binary	left to right
`+ -`	binary	left to right
`&`	binary	left to right
`^`	binary	left to right

|	binary	left to right
=	binary	left to right

8.4.3.1 Cast Operators:

These are entered following a value, register name, address, or parenthesized expression. The byte, word, and dword casts truncate the value at the specified size, resulting in an unsigned integral value. The sign extension case performs a sign extension at the operands present size.

.a	address cast
.b	byte cast
.w	word cast
.l	dword cast
.s	sign extension cast

Examples:

```
45.b
$45 #69 #69 'E'
45.w
$0045 #69 #69 '.E'
45.l
$00000045 #69 #69 '...E'
```

8.4.3.2 Unary Operators:

~	bitwise NOT
+	no operation
-	change sign

Examples:

```
~1
$fe  #254  #-2  '.'

2*-1
$fe  #254  #-2  '.'
```

8.4.3.3 Dereference Operator

This is similar to the 'C' language dereference operator '*'. The operand is either an address or an integral value.

@	retrieves 4 bytes as an unsigned integral value
@.a	retrieves 4 bytes as an address
@.b	retrieves 1 byte as an unsigned integral value
@.w	retrieves 2 bytes as an unsigned integral value
@.l	retrieves 4 bytes as an unsigned integral value

Examples:

```
@.l(A2)
```

```
$00040000  #262144  #262144  '....'

@.b(PC)
$70  #112  #112  'p'

@A7
$00000000  #0  #0  '....'
```

8.4.3.4 Binary Arithmetic Operators

*	multiplication
/	division
+	addition
-	subtraction

8.4.3.5 Binary bitwise operators

&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

8.4.3.6 Assignment operator

=	Assignment. This operator is used to change the value of any one of the processor's registers.

Example:

```
reg
D0 = 00000102 A0 = 10C0EEF6 USP = 420024FD
D1 = 00000013 A1 = 10C0EF0E SSP = 000148F4
D2 = 0000000D A2 = 000149D0
D3 = 00000000 A3 = 00015404
D4 = 00014B06 A4 = 10CF26B0
D5 = 00000000 A5 = 00013A16
D6 = 00D1F876 A6 = 000149BC PC = 10C0EEFE
D7 = 0001515E A7 = 000148F4 SR = tSxnzvc Int = 0
'SysHandleEvent'
+$0512 10C0EEFE *MOVEQ.L #$01,D0     | 7001

d0=45
$00000045 #69 #69 '...E'
```

8.4.3.7 Constants

Numbers may be entered as character constants, binary, hexadecimal, or decimal values.

A character constant is a string of one or more characters within single quotes. C-style escape sequences are supported. The result is an unsigned integral value of size determined as follows:

> 2 chars	dword
> 1 char	word
1 char	byte

Examples:

```
'xyz1'
$78797a31  #2021227057  #2021227057  'xyz1'

'a\'Y\''
$61275927  #1629968679  #1629968679  'a'Y''

'\123'
$53  #83  #83  'S'
```

A binary number is entered as a % (percent sign) followed by one or more binary digits. The size is determined as follows:

> 16 digits	dword
> 8 digits	word
1-8 digits	byte

Example:

```
%00111000
$38 #56 #56 '8'
```

A hexadecimal number is entered as one or more hexadecimal digits, or optionally a $ (dollar sign) followed by one or more hexadecimal digits. The size of the result is determined as follows:

> 4 digits	dword
> 2 digits	word
1-2 digits	byte

Example

```
c123
$c123  #49443  #-16093  '.#'

$c123
$c123  #49443  #-16093  '.#'
```

8.4.3.8 Special Variables

The processor registers can be used as variables in any expression. The data registers are named *d0* thru *d7*, the address registers are *a0* thru *a7*, the program counter is *pc*, the status register is *sr*, and the stack pointer *a7*, (which can also be referenced using it's alias name, *sp*.)

By default, any string that can represent a register name is interpreted as a register name and not a hexadecimal value. To force the string to be interpreted as a hexadecimal value, either prepend it with a 0 or a '$' character:

Example:

```
dm a0          <= display memory at address stored in register A0
10C0EEF6: 11 FC 00 01 01 01 4E 48 70 01 60 00 01 7E 4E 4F "......NHp.`..~NO"

dm a0+d0       <= Add contents of register A0 to register D0 and
                      use that as the address
10C0EFF8: 60 04 38 3C 02 07 4A 44 66 26 48 6E FF FA 48 6E "`.8<..JDf&Hn..Hn"

dm a0+0d0      <= This shows how to add the hex value 0xD0 instead of
                     the contents of register D0
10C0EFC6: 26 3C 73 79 6E 63 7A 09 60 2E 26 3C 73 79 6E 63 "&<syncz.`.&<sync"
```

8.4.3.9 Special Characters:

The following special characters are recognized in any expression:

.	last address entered
:	starting address of the current routine

Examples:

```
reg
D0 = 00000001 A0 = 10C0EEF6 USP = 420024FD
D1 = 00000013 A1 = 10C0EF0E SSP = 00014854
D2 = 0000001B A2 = 0000201C
D3 = 000020AA A3 = 0000201C
D4 = 00000000 A4 = 00001A04
D5 = 00000002 A5 = 00013A16
D6 = 00000000 A6 = 00014890 PC = 10C47284
D7 = 0001515E A7 = 00014854 SR = tSxnzvc Int = 0
'FrmDoDialog'
+$006A 10C47284 *TST.B D0                          | 4A00

dm sp
00014854: 00 01 48 78 00 00 20 AA 00 01 49 30 55 00 00 00 "..Hx.. ...I0U..."

dm .
00014854: 00 01 48 78 00 00 20 AA 00 01 49 30 55 00 00 00 "..Hx.. ...I0U..."

dm .+10
00014864: 1A 04 00 00 00 00 00 00 00 00 00 20 1C 00 00 "........... ..."

il pc
'FrmDoDialog 10C4721A'
+$006A 10C47284 *TST.B D0                                | 4A00
+$006C 10C47286 ADDQ.W #$04,A7                           | 584F
+$006E 10C47288 BNE.S FrmDoDialog+$0092 ; 10C472AC       | 6622
+$0070 10C4728A PEA -$001A(A6)                           | 486E FFE6
+$0074 10C4728E PEA -$0018(A6)                           | 486E FFE8
+$0078 10C47292 CLR.L -(A7)                              | 42A7
+$007A 10C47294 _MenuHandleEvent ; $10C4B768             | 4E4F A1BF
+$007E 10C47298 TST.B D0                                 | 4A00
+$0080 10C4729A LEA $000C(A7),A7                         | 4FEF 000C
+$0084 10C4729E BNE.S FrmDoDialog+$0092 ; 10C472AC       | 660C

il :
```

```
'FrmDoDialog 10C4721A'
+$0000 10C4721A LINK A6,-$0030                              | 4E56 FFD0
+$0004 10C4721E MOVEM.L D3/A2,-(A7)                         | 48E7 1020
+$0008 10C47222 MOVE.L $0008(A6),A2                         | 246E 0008
+$000C 10C47226 MOVE.B #$01,-(A7) ; '.'                     | 1F3C 0001
+$0010 10C4722A PEA -$0030(A6)                              | 486E FFD0
+$0014 10C4722E _FrmActiveState ; $10C48380                 | 4E4F A33B
+$0018 10C47232 MOVE.L A2,-(A7)                             | 2F0A
+$001A 10C47234 _FrmSetActiveForm ; $10C45CC8               | 4E4F A174
+$001E 10C47238 BTST #$0005,$002A(A2)                       | 082A 0005 002A
+$0024 10C4723E LEA $000A(A7),A7                            | 4FEF 000A

il :+60
'FrmDoDialog 10C4721A'
+$0060 10C4727A BNE.S FrmDoDialog+$003E ; 10C47258          | 66DC
+$0062 10C4727C PEA -$0018(A6)                             | 486E FFE8
+$0066 10C47280 _SysHandleEvent ; $10C0E9EC                | 4E4F A0A9
+$006A 10C47284 *TST.B D0                                  | 4A00
+$006C 10C47286 ADDQ.W #$04,A7                             | 584F
+$006E 10C47288 BNE.S FrmDoDialog+$0092 ; 10C472AC         | 6622
+$0070 10C4728A PEA -$001A(A6)                             | 486E FFE6
+$0074 10C4728E PEA -$0018(A6)                             | 486E FFE8
+$0078 10C47292 CLR.L -(A7)                                | 42A7
+$007A 10C47294 _MenuHandleEvent ; $10C4B768               | 4E4F A1BF
```

8.5 The Source Window

The Source window is an output-only window used for source-level debugging. As you single step, for example, the Source window follows along and displays the source code and line number corresponding to the current program counter. You can also select a specific line in the Source window for setting or clearing a breakpoint or for disassembling code. Currently, the source-level debugging support only works with code built using the GNU gcc compiler for Palm OS, although other symbol file formats may be supported in the future.

The Source window works in conjunction with the Debugger and CPU Registers windows. For example, when you single step by entering commands in the Debugger window, the Source window will automatically track along and any breakpoints set or cleared from the Debugger window are displayed in the Source window as well. There are also dedicated menu commands and key equivalents for source-level debugging that don't require you to enter commands in the Debugger window.

The Source window is split into two panes. Between the two panes is a thick horizontal line that is colored red when the device is halted in the debugger and green when the device is running code. The upper pane is used to display the values of the local variables for the current routine and the lower pane is used to display the actual source code. By default, the source pane is automatically updated every time you single step in order to show the current source file and line number corresponding to the program counter. You can also scroll the source pane or change to a different source file altogether for purposes of setting a breakpoint or just for viewing. The left margin of the lower pane is used to display indicators for breakpoints and the current program counter. Breakpoints show up with a solid red circle next to the line with the current program location indicated by a green arrow.

The source-level debugging is designed to support debugging of any type of executable code including applications, shared libraries, system extensions, or interrupt handlers. In addition, any number of executables can be source-level debugged simultaneously by loading multiple symbol files into the debugger. You can, for example, source-level debug an application and a shared library that it uses at the same time.

In order to source-level debug an executable, you must first associate a symbol file with the code for the executable that's loaded onto the device. This simply means telling the debugger the starting address of the code on the device and the name of the symbol file on the desktop that contains the symbol information for that code (there are simple menu commands for doing this). Any number of symbol files can be loaded into the debugger at once and whenever the device stops in the debugger stub for any reason, PalmDebugger will automatically determine which symbol file corresponds to the current program counter, and display the appropriate source file and line number if one was found.

A quick example helps to illustrate how this works. In order to source-level debug an application, you could do the following:

1. Ensure that the console stub is launched on the device by entering the `Shortcut-.-2` sequence.
2. Select the **Install Database and Load Symbols** menu command from the **Source** menu.
3. From the open file dialog, select the *.prc* file to load onto the device.
4. PalmDebugger now imports that *.prc* file onto the device and looks in the same directory for an associated symbol file. It then automatically associates that symbol file with the address in memory of where the application was just installed.
5. At this point, the symbol file for that application is loaded into the debugger. Any time the debugger breaks in code belonging to that application, the source window will display the associated source file and line number. Alternately, you can break into the debugger manually (using either `Shortcut-.-1` or the **Break** command from the source menu") and set a breakpoint on a certain source line of the application

using the **Toggle Breakpoint** menu command from the **Source Menu** or from the right-mouse button context menu of the source window.

If the executable you wish to debug is already loaded onto the device or is in ROM, then steps 2 thru 4 above can be replaced by selecting the **Load Symbols**... command from the **Source** menu and selecting the symbol file for that executable.

8.5.1 How Symbol Files Are Used

This section briefly describes what is contained in a symbol file and how PalmDebugger uses that information. It is recommended that you read it so you can better understand and utilize the source-level debugging support and its various features.

A symbol file is created by the linker and represents one or more code resources. Most applications for Palm OS contain only a single code resource that has a resource type of 'code' and a resource ID of 1. More complex applications may have more than one code resource and possibly more than one symbol file, if stand-alone code resources are used. Within the symbol file are names of each of the source files that were linked together to create the code resource along with the offset from the start of the code resource to the object code for each source file and each line within the source file. Also within the symbol file are descriptions of the various data structures used and the names, types, and locations of each of the local variables for each routine, as well as the global variables.

Besides the symbol file, the only other information required by PalmDebugger is the address of the code resource on the device that corresponds to that symbol file. The **Load Symbols...** and the **Install Database and Load Symbols** menu commands perform the task of lining up a symbol file with the address of the associated code resource on the device.

8.5.2 The Load Symbols Menu Commands

There are two menu commands for loading source-level symbols. The **Load Symbols for current Program Counter...** command, which is only enabled when the device *is* halted in the debugger, and the **Load Symbols...** command, which is only enabled when the device *is not* halted in the debugger.

If you select the **Load Symbols for current Program Counter...** command, the debugger first attempts to identify which code resource and database the program counter is currently in (this information can be obtained manually using the "wh \a <address>" command in the Debugger window, where <*address*> is the current program counter). Once the debugger identifies the database and code resource, it presents an open file dialog and asks the user to choose the corresponding symbol file named *"<DatabaseName>.<resType>.<resID>*.sym". If, for example, the program counter was found in the 'code' #1 resource of a database named *Text to Hex*, it asks the user to choose the symbol file *Text to Hex.code.1.sym*.

If you select the **Load Symbols...** command, then you are immediately presented with an open file dialog asking you to first select a symbol file. After you select a symbol file, the debugger looks up the address of the associated database code resource on the device and "lines up" the symbol file with that address. Note that the <*DatabaseName*> portion of the symbol file name must correspond exactly to the Palm OS name of the database, which is the name that shows up when doing a directory listing of databases on the device (using the "dir" command of the Console window) and is not always the same as the name of the *.prc* file for that database.

Finally, the **Install Database and Load Symbols...** command is a macro-type command that does two things: it imports a *prc* file into the device, then automatically loads the associated symbol file for that *prc* file. The same results can be obtained by manually importing the *prc* file using the **import** command on the Console window, then loading the symbols using the **Load Symbols...** menu command. When you choose the **Install Database and Load Symbols...** menu command, you will be presented with an open file dialog asking you to pick a *prc* file. After importing this *prc* file into the device, PalmDebugger automatically looks in the same directory for a file named <*DatabaseName*>*.code.1.sym* and associates this symbol file with the newly imported database.

8.5.3 The Source Menu

The **Source** menu contains commands for source-level debugging and for loading and releasing symbol files. The **Load Symbols..., Load Symbols for current Program Counter...,** and **Install Database and Load Symbols...** commands were already covered in the previous section. The **Remove all Symbols** command will unload all symbol files which are currently loaded. Once a symbol file is loaded, the remaining commands in the menu can be used for setting source-level breakpoints, stopping, continuing, etc.

The **Break** command is only enabled when the device is not already halted in the debugger. This command simulates entering the `Shortcut-.-1` sequence on the device and is usually more convenient to use than the equivalent Graffiti sequence. This command only works when the console stub is running on the device, however, since it relies on the console communication in order to send a key event to the device.

The **Step Into** and **Step Over** commands work at the source-level. Both commands single step one source line at a time. The difference between the two is that **Step Into** command steps into a subroutine if one is about to be called, whereas **Step Over** doesn't stop until it reaches the source line after the subroutine call.

The **Go** command continues execution and is the same as entering the "`g`" command in the Debugger window. The **Go Till** command sets a temporary breakpoint at the currently selected line in the source window and then continues execution.

The **Toggle Breakpoint** command toggles a breakpoint at the currently selected line in the source window. The **Disassemble at Cursor** command disassembles code at the currently selected line in the source window. The output of this command will appear in the Debugger window.

Finally, the **Show Current Location** command automatically scrolls the source window to show the current source file and line number. This is useful if you've previously scrolled the source window to set or clear a breakpoint or temporarily changed it to view a different source file (using the context menu, as described below).

The Source Window Context Menu

The source window has a context menu that can be activated by right-clicking with the mouse. This menu has many of the same commands that also appear in the **Source** menu of the menu bar including **Break**, **Go Till**, **Toggle Breakpoint**, **Disassemble at Cursor**, and **Show Current Location**.

Also present in the context menu are pop-up items for selecting which source file to view. Every symbol file that is loaded presents a pop-up that lists the source files for that symbol file. Using these menus, you can change the current source file for viewing purposes or for setting and clearing breakpoints.

Limitations

Unfortunately, the source-level support in PalmDebugger is still quite limited compared to most modern source-level debuggers. It provides the bare essentials for source-level debugging, and you'll find that there are numerous occasions in which you will have to switch to the assembly level Debugger window and enter commands there for certain functions:

1. The source window does not display a current stack crawl. To get a stack crawl, you must enter the `sc` command in the Debugger window.

2. Local variables are only displayed in hexadecimal format.

3. The values of local variables cannot be changed from the source window. The only way to change them is to use the `sb`, `sw`, or `sl` commands from the Debugger window and you must enter the address of the variable manually.

8.6 Debugging Hints

This section describes a few hypothetical debugging situations. These examples are included to help familiarize you with the debugging commands and how they can be used together to track down certain types of problems.

8.6.1 Entering the Debugger

The most common way to enter the debugger is to enter the Graffiti sequence `Shortcut-.-1`, i.e., the shortcut stroke followed by two taps to generate a period, followed by the number 1 (in the right side of the Graffiti area).

If you have already started the console stub on the device (using the `Shortcut-.-2` sequence), you can use the **Break** command in the **Source** menu. This command sends a key event to the device using the console stub communications and this key event is identical to entering the `Shortcut-.-1` Graffiti sequence by hand.

You can also rebuild your executable with a compiled breakpoint in it by making a call to `DbgBreak()`. Remember, though, that unless you've entered the debugger at least once already using the `Shortcut-.-1` sequence, a `DbgBreak()` call will display a fatal error dialog instead of placing you in the debugger.

Finally, you can enter the debugger immediately after reset by holding the down button and pressing the reset button in the back of the device with a paper clip. This will put you into the SmallROM debugger, which is bootstrap code placed into the very front of the ROM that contains just enough code to initialize the hardware and startup the debugger stub. If you enter the "`g`" command at this point, the system will jump to the BigROM. The BigROM contains the same code as the SmallROM as well as the rest of the system code. If you press the down button on the device while executing the "`g`" command, you will end up in the BigROM's debugger. You can now set a-trap breaks, or single-step through the boot-up sequence.

The Handspring device uses USB by default for debugger communication, whereas all other Palm devices use serial only. If desired, you can also debug a Handspring device using serial by holding the "up" key when entering the debugger for the first time. This works when entering the Graffiti shortcut sequence or during reset as well. To enter the serial debugger during reset, hold both the [Up] and [Down] keys while pressing the reset button.

8.6.2 Finding Code

A common problem is finding the location in memory at which code resides. The ultimate goal is being able to single-step through the problem code to determine exactly what is going wrong. Depending on the circumstances, you may use any one of a number of different methods.

Method 1:

If you're debugging an application and are able to rebuild it, it is sometimes easiest just to re-build the application with a `DbgBreak()` call in the problem routine. This is a "compiled" breakpoint and will cause your program to break into the debugger at that line.

Method 2:

A second method is to use PalmDebugger to set an a-trap break on a system call that the problem routine makes. Ideally, it would be a system call that *only* that routine makes so that you won't get false triggers from other routines making the same call. For example, if you wanted to find your application's main event loop, you could set an a-trap break on `EvtGetEvent()`:

```
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 you break due to an a-trap break, you end up at the beginning of the system call. At this point, the return address on the stack is the routine that actually made the system call which in this case is your application's main

event loop. To get back to this routine, you can either single-step through the `EvtGetEvent()` call until it returns (not recommended!) or set a temporary breakpoint at the return address on the stack. To do this, use the following command:

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

The '@' operator fetches the long word at the given address (the value stored in the stack pointer), so the expression '`@sp'` yields the return address on the stack.

You are now at the instruction in your main event loop immediately after the `EvtGetEvent()` call. At this point, you may want to disassemble this routine from the beginning. This is where the ':' symbol, which represents the starting address of the current routine, is handy:

```
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
il :+6c
'EventLoop 1001B2D0'
+$006C 1001B33C _FrmDispatchEvent ; $10C4769A       | 4E4F A1A0
+$0070 1001B340 ADDQ.W #$04,A7                      | 584F
+$0072 1001B342 CMPI.W #$0016,-$0018(A6) ; '..'     | 0C6E 0016 FFE8
+$0078 1001B348 BNE.S EventLoop+$000C ; 1001B2DC    | 6692
+$007A 1001B34A MOVEM.L -$0028(A6),D3-D4/A2         | 4CEE 0418 FFD8
+$0080 1001B350 UNLK A6                             | 4E5E
+$0082 1001B352 RTS                                 | 4E75 8945 7665 6E74
dm 1001b354
1001B354: 89 45 76 65 6E 74 4C 6F 6F 70 00 00 4E 56 00 00 ".EventLoop..NV.."
```

Note the '*' to the left of the instruction after the `_EvtGetEvent` call - this marks the current location of the program counter. Also notice that PalmDebugger was able to tell the name of the routine without your having to load a symbol file. This is possible because the name of the routine is included in the code itself by the compiler. If you display memory immediately after the return instruction of a routine, you can see the name of the routine embedded in the code:

Another way to double-check that you are where you want to be, is to perform a stack crawl. The routines are displayed from "oldest" at the top to "newest" at the bottom.

```
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
```

Yet another check is to get a list of the opened databases. If you are in the application you expect to be, it should appear as one of the opened databases. Note that the "System" and "Graffiti ShortCuts" databases are always opened by the system and always appear at the bottom of this list.

```
opened
```

```
name                      resDB cardNum  accessP        ID             openCnt mode
-------------------------------------------------------------------------------
Tex2HexDB                 no     0       00015146       0001825B       1       0003
*Text to Hex              yes    0       00016DD2       0001821B       1       0001
*Graffiti ShortCuts       yes    0       00017D5C       0001812B       1       0007
*System                   yes    0       00017FEE       00D20A44       1       0005
-------------------------------------------------------------------------------
Total: 4 databases opened
```

Method 3:

This brings us to the third way to find your code… by the name of the routine. The `'ft'` command in the Debugger window will find text, and if your routine name is fairly unique it is an easy way to find your routine. The `'ft'` command takes 3 arguments: the text to find, a starting address, and the number of bytes to search. To search the first megabyte of RAM on a Visor or Palm III for example, use this command:

```
ft "EventLoop" 00000000 100000
dm 100005C4 ;100005C4: 45 76 65 6E 74 4C 6F 6F 70 63 61 74 69 6F 6E 00 "EventLoopcation."

After the above line is printed, hit Enter again without parameters to repeat the find
dm 1001B355 ;1001B355: 45 76 65 6E 74 4C 6F 6F 70 00 00 4E 56 00 00 2F "EventLoop..NV../"
```

The 'ft' command stops at the first successful find of the text and continues the search from where it left off if you hit *Enter* again without any parameters. All existing PalmPilots except for the PalmV have RAM starting at address 0x10000000. The PalmV and Visor RAM starts at address 0. To search ROM instead of RAM, use address 0x10C00000 for early Visor or Palm devices, or 0x10000000 for later models. Use ‘dl0’ to find the dynamic heap storage RAM, and ROM listed as heaps 0, 1 and 2 respectively. Note that the first occurrence of the text was found at 0x100005C4. This is actually an alias of the debugger globals that are stored in low memory at address 0x05C4 and is a copy of the string you asked the debugger to search for and not the actual routine. To ensure that the address is of the routine you want, disassemble code before that address:

```
il 1001b355-23
'EventLoop 1001B2D0'
+$0062 1001B332 _FrmSetEventHandler ; $10C47672     | 4E4F A19F
+$0066 1001B336 ADDQ.W #$08,A7                      | 504F
+$0068 1001B338 PEA -$0018(A6)                      | 486E FFE8
+$006C 1001B33C _FrmDispatchEvent ; $10C4769A       | 4E4F A1A0
+$0070 1001B340 ADDQ.W #$04,A7                      | 584F
+$0072 1001B342 CMPI.W #$0016,-$0018(A6) ; '..'     | 0C6E 0016 FFE8
+$0078 1001B348 BNE.S EventLoop+$000C ; 1001B2DC    | 6692
+$007A 1001B34A MOVEM.L -$0028(A6),D3-D4/A2         | 4CEE 0418 FFD8
+$0080 1001B350 UNLK A6                             | 4E5E
+$0082 1001B352 RTS                                 | 4E75 8945 7665 6E74
```

Note that we took the address of the found text (0x1001b355) and subtracted 23 bytes. We chose an odd number because all instructions must be on word boundaries and 0x1001b355 is an odd address.

Method 4:

The last (and sometimes easiest) method to find your code is to take advantage of the source-level debugging support. Assuming you've built your application with the gcc compiler and have generated a symbol file, load the symbol file using the **Load Symbols**... menu command. Remember to launch the console stub on the device first or the **Load Symbols...** command will not work. Once the symbol file is loaded, break into the debugger stub on the device using the **Break** command of the Source menu, select the source line in the **Source** window of PalmDebugger, and set a breakpoint there using the **Toggle Breakpoint** command from the **Source** menu or from the source window's context menu.

8.6.3 Finding Memory-Trashing Bugs

Memory-trashing bugs are often the hardest kind to track down. A bug in the code could trash low memory globals used by the operating system, the dynamic memory heap, or an application variable. A memory corruption in any one of these areas could cause very unpredictable behavior.

The first line of attack on these kinds of bugs is to divide and conquer: using forms of binary search, try and narrow down which portion of the code is corrupting memory.

Heap Corruptions

If a bug trashes a memory heap, you might get a fatal error message put up by the Memory or Data Managers or you may simply crash on some unrelated manager due to a side effect. In any case, if you do suspect a corrupted heap, use the 'hd 0' command to dump the dynamic heap. If the heap is in a valid state, you'll see the heap dump complete and print out totals at the bottom of the heap for amount of memory-used, memory-free, and other statistics:

```
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 00001490 F0001491 00001E 000026  #0  #0    fM Alarm Table
-0000155A 00001494 F0001495 000456 00045E  #0  #0    fM Graffiti Private
-000019B8 00001498 F0001499 000012 00001A  #0  #0    fM DataMgr Protect List
(DmProtectEntryPtr*)
...
-00017DC6 -------- F0017DC6 0001F4 0001FC  #0 #15    fM Handle Table: 'Graffiti ShortCuts'
-00017FC2 -------- F0017FC2 000024 00002C  #0 #15    fM DmOpenInfoPtr: 'Graffiti ShortCuts'
-00017FEE -------- F0017FEE 00000E 000016  #0 #15    fM DmOpenRef: 'Graffiti ShortCuts'
-------------------------------------------------------------------------------------------
Heap Summary:
  flags:              8000
  size:               016B80
  numHandles:         #40
  Free Chunks:        #14     (010A90 bytes)
  Movable Chunks:     #52     (006040 bytes)
  Non-Movable Chunks: #0      (000000 bytes)
```

For a faster check of the heap without a heap dump, use the 'hchk' command. This simply checks the validity of the heap:

```
hchk 0
Heap OK
```

By breaking into the debugger at various portions of execution and checking the heap using either hd or hchk, as shown above, you can narrow down the heap corruption.

Another method for tracking down heap corruption is to use the `mdebug` command in the Console window. This command is one of the Console commands that can be executed even when the debugger is attached. (See *Using the Console Window When the Debugger is Attached* below for more info). This command puts the device into one of a set of various heap checking modes. Basically, when the device is in this mode, it performs an automatic heap check and verification on every memory manager call. If it detects a heap corruption during any of these checks, it automatically breaks into the debugger. Type `help mdebug` to get a full list of options for this command. By turning various memory checking features on and off, you can usually strike an acceptable balance between performance (various modes can slow down performance considerably) and coverage.

```
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
```

Global Variable Corruptions

Some bugs may trash a global - either a low memory system global or an application global. The effects of this type of corruption are generally unpredictable and the process of tracking them down is quite difficult and hard to generalize.

Once you have managed to determine which address in memory is being corrupted, however, you can usually use the 'ss' (Step-Spy) command to help determine where the bug in the code is. The 'ss' command puts the processor in a single-step mode in which it will automatically check the contents of a given address after every instruction, and automatically break into the debugger if the contents ever change. It takes one parameter which is the address of a DWord in memory to check.

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

Because the 'ss' command makes the processor single-step through instructions, it makes the device run considerably slower, so you should usually narrow the bug down to a certain area before using this command.

8.6.4 Viewing Local Variables and Function Parameters

If you are debugging using the source-level window, the local variables and parameters for functions are displayed in the upper pane of the window. If you don't have access to symbol information, however, you have to manually look up the variable values using commands in the Debugger window. This section walks through how to do this with a typical function.

To illustrate the process, we will use the following example routine:

```
static Boolean
MainFrmEventHandler (EventPtr eventP)
{
  FormPtr      formP;
  Boolean      handled = false;
  Err          err;
  char         buffer[64];
  DWord               bytes=0;
  SWord               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 we were to break into the debugger at the beginning of the routine, (right before it calls `StdHandleEvent()`), you would see the following disassembly:

```
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
```

When a routine enters, the first DWord on the stack is the return address. Following that are the parameters, from left to right. If, for example, we were to display memory to which the stack pointer points on the first instruction of the routine (the LINK instruction), we would 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 DWord (0x10C47700) is the return address of the routine. The second DWord (0x00014A4E) is the `eventP` parameter to the routine. If there were another parameter to the routine, it would follow the `eventP` parameter on the stack.

After the LINK instruction executes, however, the stack pointer register is changed. This happens because the stack pointer is decremented to make room for local variables (in this example, to make room for 0x48 bytes of local variables) and for a saved value of the A6 register. The A6 register is changed to point the beginning of the function's stack frame. The A6 register then acts like a stack frame pointer and is used by the rest of the routine to access function parameters and local variables. Here's a picture of what the stack looks like after the LINK instruction:

```
    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
```

Thus, if you display memory at register A6, you will 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 DWord that A6 points to is the old value of A6. The next DWord is the return address of the routine, followed by the parameters to the function. Thus, the first parameter to the function can always be found at 8(A6) and indeed if you look at the function disassembly above, you can see that one of the first things it does is after the LINK instruction is copy the value of the 'eventP' parameter from 8(A6) to register A2.

The function local variables are placed at memory locations before A6. For example, the `bytes` local variable, which is a DWord, is found at -$0044(A6) and is cleared to 0 by the instruction at address 0x1001E2A4. It is not always easy to tell where each local variable is on the stack except by disassembling the code and looking for places in the code where that variable is accessed. Once you know the offset of the variable, you can view by using an expression with A6. To view the value of the `bytes` parameter, for example, you could use the following command:

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

8.6.5 Using the Console Window When the Debugger is Attached

When the device is in the debugger stub and PalmDebugger is attached, you normally use the Debugger window only to enter commands. There is, however, a subset of console commands that work even when the debugger is attached. The subset of console commands that work when the debugger is attached are those that simply display information and don't change memory contents. These include 'dir', 'hl', 'hd', 'hchk', 'mdebug', and others.

When the debugger is attached, commands that you enter in the Console window do not go through the normal channels and talk to the console stub on the device. They instead talk directly to the debugger stub so that they can be executed even when the console stub has not been started.

By leveraging the use of these commands in the Console window, you now have two windows at your disposal for displaying debugging information. This comes in handy for displaying a heap dump in the Console window that you can view while single-stepping in the Debugger window.

8.6.6 Changing PalmDebugger's baud rate

Whenever the debugger stub or console stub on the device starts up, it starts out communicating at 57,600 baud. It is often desirable to switch to a higher baud rate in order to download large applications or other files to the device or to a lower baud rate if you are using a serial cable without hardware handshaking lines.

If the device is in the debugger stub, and PalmDebugger is attached to the device, you can change the baud rate of both PalmDebugger and the device at the same time by choosing a new baud rate setting from the **Connection** menu of PalmDebugger. When PalmDebugger thinks the device is attached, it sends a request packet to the device's debugger stub telling it the new baud rate to use. It then it switches over to the new baud rate itself. Keep in mind that this new baud rate setting only remains in effect until the device is soft-reset, or until you launch an application on the device that opens the serial port and changes the baud rate again.

You can also change the baud rate of both PalmDebugger and the device through the console stub. If you change PalmDebugger's baud rate when it is *not* attached to the debugger stub, it sends the new baud rate request to the console stub on the device instead of the debugger stub. Whether you change the baud rate through the console stub or the debugger stub, the other stub will use the new baud rate as well.

8.6.7 Debugging Applications That Use the Serial Port

Debugging applications that use the serial port when using that same serial port for the debugger is tricky, but not impossible. As long as the application itself and the debugger stub are using the same baud rate, you can actually perform limited debugging functions. When the debugger stub starts up, it always initializes the serial port on the device to 57,600 baud and assumes it will stay at that baud rate unless it gets a message from PalmDebugger while attached telling it to change (as a result of the user picking a new baud rate from the **Connection** menu).

If the debugger stub on the device has been entered at least once already, and you later launch an application that opens the serial port, that application may change the baud rate. If it does, the debugger stub on the device will end up using that new baud rate the next time it enters. If this is the case, you will have to manually change the baud rate setting on the PalmDebugger application's **Connection** menu in order to communicate again with the device.

Of course, when you do enter the debugger stub on the device while debugging a serial application, the debugger stub will send data over the serial port and most likely disrupt serial communications with whatever host your Palm OS application was originally talking to. You can, however, switch the serial cable back over to PalmDebugger, double-check your baud rate setting, and issue an 'att' command to attach to the device and perform "post-mortem" analysis at that point.

What you *cannot* do when using a serial port application on the device is use the console stub. When the console stub starts up, it opens up the serial port like a normal application would, and thus prevents any other application from successfully opening up the port at the same time. Likewise, if you are in an application that has the serial port open, you will not be able to start up the console stub. Remember that a number of console commands can be used even when the device is in the debugger stub. This subset of console commands (those that merely display information about heaps and databases) are smart enough to know when the device is currently in the debugger stub, and can communicate with that stub instead of the console stub.

8.6.8 Importing System Extensions and Libraries

The console's 'import' command imports a new database or replaces an existing database on the device. It can only replace an existing database, however, if that database is not currently open or protected. System extension databases and shared libraries present a problem because they are normally kept open or marked protected so that they won't be deleted accidentally while they are in use by the system.

If you are developing a system extension or shared library and need to import a newer version to the device, you must make sure the old database is closed and unprotected first. If it is not, you will get the following error from the import command:

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

To get around this, soft-reset the device while pressing the [Up] button on the device. The [Up] button tells the system that it should not automatically load system extensions or shared libraries. You can then import your database and soft-reset again to make the system use the new version.