Device Abstraction Layer (DAL) Customization Guide

Palm OS® 5 PDK/SPK

PalmSource Confidential

Written by Anna Schaller, Jenny Green, and Mark Dugger
Engineering contributions by Clif Liu, Ken Krugler, Vivek Magotra, and Steve Lemke.

Copyright © 1996-2004, PalmSource, Inc. and its affiliates. All rights reserved. This technical documentation contains confidential and proprietary information of PalmSource, Inc. (“PalmSource”), and is provided to the licensee (“you”) under the terms of a Nondisclosure Agreement, Product Development Kit license, Software Development Kit license or similar agreement between you and PalmSource. You must use commercially reasonable efforts to maintain the confidentiality of this technical documentation. You may print and copy this technical documentation solely for the permitted uses specified in your agreement with PalmSource. In addition, you may make up to two (2) copies of this technical documentation for archival and backup purposes. All copies of this technical documentation remain the property of PalmSource, and you agree to return or destroy them at PalmSource’s written request. Except for the foregoing or as authorized in your agreement with PalmSource, you may not copy or distribute any part of this technical documentation in any form or by any means without express written consent from PalmSource, Inc., and you may not modify this technical documentation or make any derivative work of it (such as a translation, localization, transformation or adaptation) without express written consent from PalmSource.

PalmSource, Inc. reserves the right to revise this technical documentation from time to time, and is not obligated to notify you of any revisions.

THIS TECHNICAL DOCUMENTATION IS PROVIDED ON AN “AS IS” BASIS. NEITHER PALMSOURCE NOR ITS SUPPLIERS MAKES, AND EACH OF THEM EXPRESSLY EXCLUDES AND DISCLAIMS TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, ANY REPRESENTATIONS OR WARRANTIES REGARDING THIS TECHNICAL DOCUMENTATION, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING WITHOUT LIMITATION ANY WARRANTIES IMPLIED BY ANY COURSE OF DEALING OR COURSE OF PERFORMANCE AND ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, ACCURACY, AND SATISFACTORY QUALITY. PALMSOURCE AND ITS SUPPLIERS MAKE NO REPRESENTATIONS OR WARRANTIES THAT THIS TECHNICAL DOCUMENTATION IS FREE OF ERRORS OR IS SUITABLE FOR YOUR USE. TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, PALMSOURCE, INC. ALSO EXCLUDES FOR ITSELF AND ITS SUPPLIERS ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, EXEMPLARY OR PUNITIVE DAMAGES OF ANY KIND ARISING OUT OF OR IN ANY WAY RELATED TO THIS TECHNICAL DOCUMENTATION, INCLUDING WITHOUT LIMITATION DAMAGES FOR LOST REVENUE OR PROFITS, LOST BUSINESS, LOST GOODWILL, LOST INFORMATION OR DATA, BUSINESS INTERRUPTION, SERVICES STOPPAGE, IMPAIRMENT OF OTHER GOODS, COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR OTHER FINANCIAL LOSS, EVEN IF PALMSOURCE, INC. OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR IF SUCH DAMAGES COULD HAVE BEEN REASONABLY FORESEEN.

PalmSource, the PalmSource logo, BeOS, Graffiti, HandFAX, HandMAIL, HandPHONE, HandSTAMP, HandWEB, HotSync, the HotSync logo, iMessenger, MultiMail, MyPalm, Palm, the Palm logo, the Palm trade dress, Palm Computing, Palm OS, Palm Powered, PalmConnect, PalmGear, PalmGlove, PalmModem, Palm Pack, PalmPak, PalmPix, PalmPower, PalmPrint, Palm.Net, Palm Reader, Palm Talk, Simply Palm and ThinAir are trademarks of PalmSource, Inc. or its affiliates. All other product and brand names may be trademarks or registered trademarks of their respective owners.

IF THIS TECHNICAL DOCUMENTATION IS PROVIDED ON A COMPACT DISC, THE SOFTWARE AND OTHER DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENTS ACCOMPANYING THE SOFTWARE AND OTHER DOCUMENTATION.

DAL Customization Guide
Document Number 5044-004
January 27, 2004

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

Table of Contents

About This Document v

What this Guide Contains . . . v
Related Documentation . . . vi
General Documents . . . vi
Technology-Specific Documents . . . vii
Tool-Specific Documents . . . viii
Additional Resources . . . viii

1 Development Process 1

Tool Chain . . . 2
Modifying the DAL Source. . . . 3
Scatterload files . . . 3
Building <source>.prc . . . 4
SmallDAL . . . 4
BigDAL . . . 5
Serial, USB, DMA, and SD Card Drivers . . . 5
Building a ROM . . . 6
Installing the ROM into RAM . . . 7
Debugging the ROM . . . 7

2 Configuring Memory 9

Memory Regions . . . 9
Virtual Memory Layout . . . 11
Virtual Memory Map . . . 11
HwrPreRAMInit() Responsibilities . . . 15
New Page Tables. . . . 16
Low Memory Globals . . . 16
Scaling Heap Space . . . 17

3 DAL Globals 19

DAL Globals Data Structure . . . 20
HAL Globals . . . 20
Setting Globals from the Card Header . . . 28
Identifying Hardware Features . . . 30

Blitter Globals 32
Screen Manager Globals 33

4 Dynamic Input Area **35**
Input Area Resources 35
Input Area Feature Flags. 36

5 Using the Control Bar **39**
Control Bar Resources 39
Customizing the Control Bar 40
Editing the Bitmaps. 41
Adding Buttons 42
Removing Buttons 44

6 Replacing System Fonts **45**
Creating a Font Database 47
Replacing Fonts in the Locale Module 48
GenerateXRD 48
PalmRC. 49
Creating a Hierarchical Font 50
Subfonts 50
The Font Index Resource 51
Font Map Resources 53
How a Hierarchical Font Works 57
Creating Font Resources 59
Standard Font Resources 59
Extended Font Resources 63

Index **69**

About This Document

The *DAL Customization Guide* provides background, conceptual, and how-to information on the Device Abstraction Layer of the ROM image. This manual complements the API's discussed in the *DAL Reference*. It provides the common design and implementation information that is needed to port the Palm OS® to a custom hardware platform.

What this Guide Contains

- Chapter 1, "Development Process," describes the development cycle for creating and testing a custom DAL.
- Chapter 2, "Configuring Memory," describes the memory map, with emphasis on regions memory is partitioned into. Memory is configured in two passes; the first pass creates a preliminary map, the second creates a final runtime map. The events related to configuring the map during each of these passes is discussed later in this chapter, under the section on Virtual Memory Map.
- Chapter 3, "DAL Globals," describes the globals supported within the DAL, and where appropriate, which values need to be customized.
- Chapter 4, "Dynamic Input Area," describes how to customize the dynamic input area available on some Palm Powered™ handhelds.
- Chapter 5, "Using the Control Bar," describes how to customize the control bar available with some display ROMs.
- Chapter 6, "Replacing System Fonts," describes how to replace the Palm OS system fonts with your own.

Information related to specific technologies can be found in the relevant technology manual.

Related Documentation

The documents in the following sections make up the Palm OS 5 PDK and SPK documentation set. The following manuals should be used in conjunction with the *DAL Customization Guide*. Each one provides information that correlates to information described in this manual.

- General Documents
- Technology-Specific Documents
- Tool-Specific Documents

These documents can be found in the `Development Kit\Documentation` directory on the Palm OS 5 PDK and Palm OS 5 SPK, unless otherwise noted.

General Documents

Document	Description
Introduction to the PDK or *Introduction to the SPK*	Guide that orients you to all the kits, tools, code, and documentation on the PDK (Product Development Kit) or SPK (Silicon Porting Kit).
Coding Recommendations	This document contains a set of recommendations that make the code you write easier for other developers to maintain. It is not a set of coding guidelines, but rather suggests ways of solving common development problems.
Shared Library Design Guide	This manual provides information on customizing Palm OS using ARM-native code. Discussion includes writing ARM shared libraries, integrating ARM code with 68K applications, and creating patches.
Architectural Overview	The manual provides background and conceptual information on the design of Palm OS 5. Information related to specific technologies can be found in the relevant technology manual.

Document	Description
DAL Reference	This manual is a companion to the *DAL Customization Guide*. It describes the API routines in the Hardware Abstraction Layer (HAL), the kernel Hardware Abstraction Layer (kHAL), and the Kernel Abstraction Layer (KAL) . These routines serve two purposes. They are either modifed by you to accomodate specific hardware features, or called to accomplish a particular task.
Glossary of Terms	This document contains the master glosssary of terms used in the Palm OS 5 PDK/SPK documentation.

Technology-Specific Documents

Document	Description
Display Driver Design Guide	Technology guide on creating a hardware-specific display driver that communicates with the screen manager and the blitter routines.
Ethernet Interface Design Guide	Technology guide to implementing an Ethernet interface on the Palm OS. This document is particularly relevant to those implementing wireless Ethernet interfaces such as IEEE 802.11b.
Expansion Manager Solutions Guide	Technology guide that provides you with background information and instruction on extending Palm OS to include expansion slot technology. The information in this guide builds on the Expansion Manager and VFS Manager chapters in the *Palm OS Programmer's Companion* and *Palm OS Programmer's API Reference*.
Serial Communications Driver Design Guide	Technology guide on writing virtual communication drivers. Supported drivers include serial as well as USB.
Sound Driver Design Guide	Technology guide to creating a hardware-specific sound driver that communicates with the Sound Manager.

PalmSource Confidential

Tool-Specific Documents

Document	Description
Building a ROM	This guide begins by providing a description of the various ROM components. It then describes the tools and steps needed to integrate the DAL, the Palm OS®, and the built-in applications into an image for installation in flash ROM or in RAM.
Building Palm OS Application Interfaces	This book describes a set of developer tools that you can use to create, edit, process, and compile Palm OS resources—forms, menus, text strings, and controls—for Palm OS applications. The Palm resource tools operate on an XRD file format rather than Macintosh resource binary format (RSRC) format that was previously used. GenerateXRD, PalmRC, and PRCMerge are the tools are used in this process.
Data Integration Tools	This manual describes three tools you can use to manipulate Palm OS databases: DDEditor, which you use to edit Palm OS database information; DDMerge, which lets you merge new data into existing Palm OS databases; and hOverlay, with which you can generate base and overlay PRCs.
Debugging a ROM	This manual provides conceptual, guidance, and reference information for developers who want to use Palm OS Debugger to debug Palm OS applications and shared libraries.
Customizing Palm OS Simulator	Guide to creating a custom version of the Palm OS Simulator. This document is located in the `Development Kit\Tools\Simulator` directory.

Additional Resources

- Documentation

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

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

- Training

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

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

- Knowledge Base

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

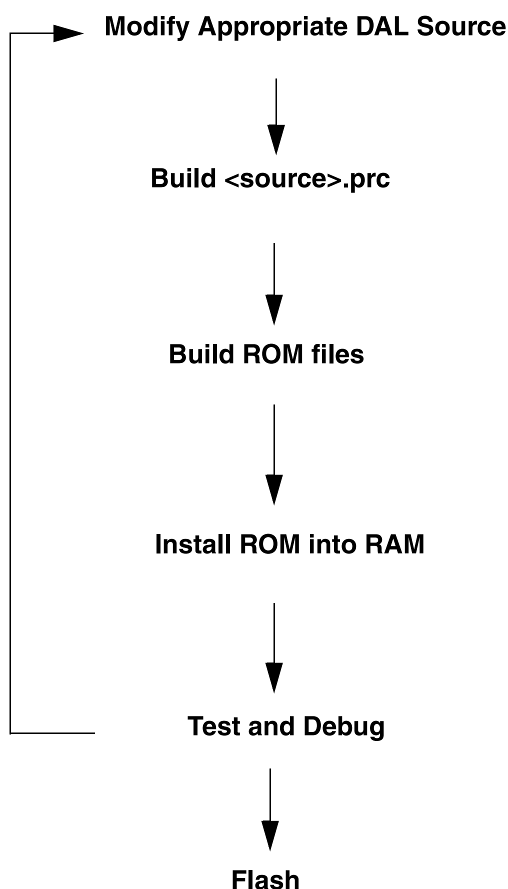
  http://www.palmos.com/dev/support/kb/

PalmSource Confidential

1

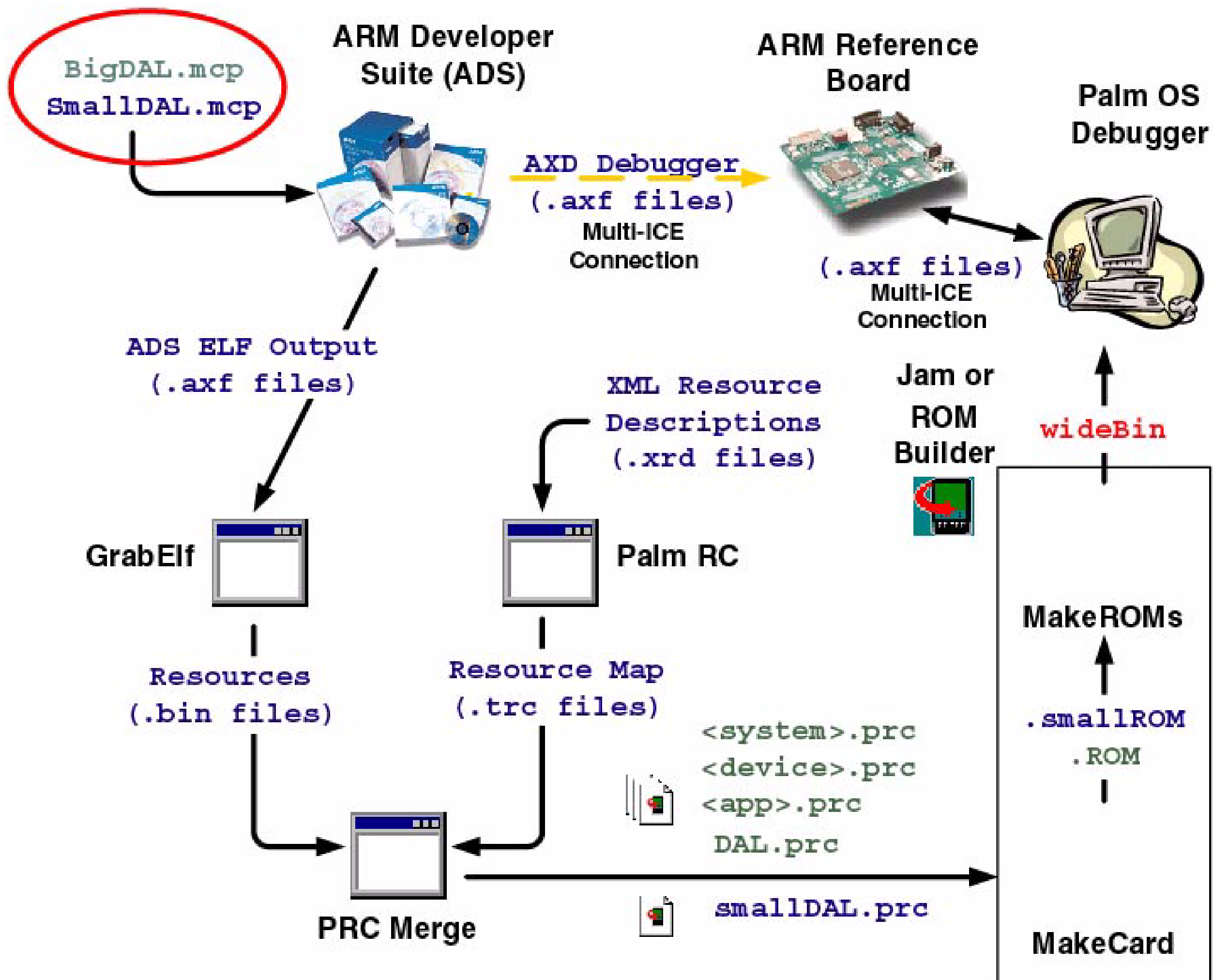
Development Process

The development process is a cycle of steps. Each cycle either adds new functionality or further debugs modified code. The process can be viewed as shown here. Note that some of the steps are performed automatically by the Palm OS® 5.4 automated build system.



Tool Chain

The tools used in the development process, as well as the files generated at each step, are shown in the diagram below. Note that you can either build a ROM using the Palm OS 5.4 automated build system which is based on Jam, or you can build a ROM using the ROM Builder tool. For details on Jam and ROM Builder, see *Building a ROM*.



It is beyond the scope of this chapter to describe these tools. This diagram serves only to show the development cycle from a tools perspective. Refer to *Building a ROM* for more information on these tools.

Modifying the DAL Source

The DAL is made up of three components: the HAL, the RAL, and the kHAL. The modifiable components include only the HAL and the kHAL.

Within the HAL, there are several components that can be modified and rebuilt as follows:

- SmallDAL — Contains the bootstrap ROM and Debugger Stub
- BigDAL — Contains the hardware access APIs used by Palm OS®, as well as the kernel static library.
- Serial Driver — Contains the hardware access code to support serial communication over an RS232 connection.
- USB Driver — Contains the hardware access code to support USB communication over a USB port.
- DMA Driver
- MX1 SD Card Driver

Scatterload files

A limitation in CodeWarrior IDE prevents having more than one file with the same name in a project. Since the scatter load files specify the file name of the intermediate file containing the compiled object code, and the OS build system uses the same file name for all entry points into the built libraries, separate scatter load descriptions have been created for each PRC. Each scatter load file is located in the same directory as the Jamfile for the relevant PRC. For example, for `SmallDal.prc`, the scatter load file (`MX1_SmallDAL.txt`) is located in the same directory as the Jamfile for `SmallDal.prc`. The scatter load file is a description file that tells the linker how to organize the resulting compiled code in the `.axf` file.

PalmSource Confidential

Building <source>.prc

The build process for the files differs slightly depending on what kind of file you are modifying. The process, from source file to PRC file, is built into the automated build system files (such as the Jamfiles and Jamrules), and no intervention is required on your part. Behind the scenes, however, there are several tools that facilitate the build that you should be aware of. The <source>.prc files are confined to the following files:

- SmallDAL.prc
- BigDAL.prc
- Serial.prc
- USB.prc
- DMALib.prc
- MX1MMC.prc

SmallDAL

The SmallDAL is the only component of the SmallROM. It contains the bootstrap code.

Once you have modified your files for the SmallDAL, you can use the Palm OS 5.4 automated build system, based on Jam, to initiate the build. (For detailed information on Jam and the automated build system, see *Building a ROM*.) Behind the scenes the following things occur:

- `.axf` files are generated.
- Post-Linker batch file runs GrabElf tool.
  - GrabElf decompiles `.axf` into separate generic `.grc` files. It then renames `.grc` files to `.bin` files with a file name that encodes its resource Type and ID.
- Post-Linker batch file runs PRCMerge tool.
  - PRCMerge creates `.prc` files from the `.bin` resource files.

BigDAL

The BigDAL contains much of the same code that is in the SmallDAL. The boot routines are repeated and re-run in the BigDAL. In addition to running the boot code again, the BigDAL also sets up the kernel and runtime environment, loads device drivers, and transfers control to the system startup—PalmOS Main.

Once you have modified your files for the BigDAL, you can use Jam to initiate the build. (For detailed information on Jam, see *Building a ROM*.) Behind the scenes the following things occur:

- `.axf` files are generated.
- Post-Linker batch file runs GrabElf tool.
  - GrabElf decompiles `.axf` into separate generic `.grc` files. It then renames `.grc` files to `.bin` files with a file name that encodes its resource Type and ID.
- Post-Linker batch file runs PRCMerge tool.
  - PRCMerge creates `.prc` files from the `.bin` resource files.

Serial, USB, DMA, and SD Card Drivers

Information on creating the Serial and USB drivers is discussed in the *Serial Communications Driver Design Guide*.

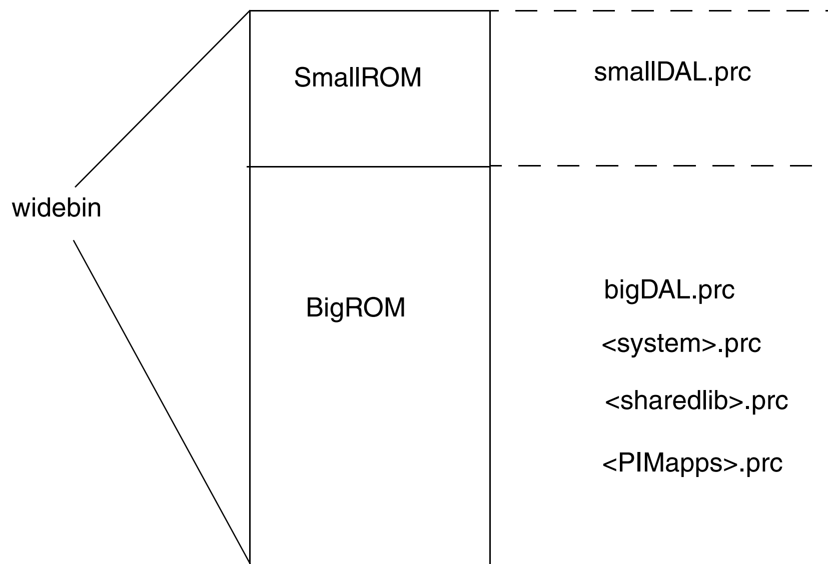
Once the you have modified the files for your drivers, you can use Jam to initiate the build. (For detailed information on Jam, see *Building a ROM*.) Behind the scenes the following things occur:

- `.axf` files are generated.
- Post-Linker batch file runs GrabElf tool.
  - GrabElf decompiles `.axf` into separate generic `.grc` files. It then renames `.grc` files to `.bin` files with a file name that encodes its resource Type and ID.
- Post-Linker batch file runs PalmRC tool.
  - PalmRC compiles XML resource descriptions into `.trc` guide file.
- Post-Linker batch file runs PRCMerge tool.

– PRCMerge creates `.prc` files from the `.trc` guide file and `.bin` resource files.

Building a ROM

Once you have the individual .prc file(s), the ROM images can be created. There are two separate ROM’s in a Whole ROM, or widebin, file. The widebin file includes the following components:



The ROM Builder tool can be used to create the widebin file, or you can use the automated build system based on Jam. Refer to the *Building a ROM* guide for detailed information. The process is initiated Jam, or by a batch file called by ROM Builder. This process is automated, and requires no intervention on your part. An understanding of what goes on behind the scenes is provided for information only.

- ROM Builder calls batch file, or you run Jam.
- Jam or batch file runs MakeCard tool
  – MakeCard combines `.prc` files into `.rom` file (card image).

- Jam or batch file runs MakeROMs tool.
  - MakeROMs combines .rom files into `.widebin` file (Whole ROM).

NOTE: Keep in mind that there are multiple output format options for use during both product development and manufacturing. These options are straight binary and Motorola S-Record.For proper tools support, all final ROM builds should have a unique HAL ID, and an appropriate company ID. The values are set in the sample batch files to PalmSource specific identifiers. These batch files must be updated to reflect your values. These values should be the same as those assigned to the GHwrOEMHALID and GHwrOEMCompanyID globals during OS boot. See the section "Setting Globals from the Card Header" in Chapter 3, "DAL Globals," for information on setting these globals.

Installing the ROM into RAM

Initial development will be focused on getting your SmallROM to a state that boots the board. The SmallROM should be modified, loaded into RAM, and tested from there. This cycle will be repeated until you get a working version. Once your SmallROM boots the board you should flash it so that you do not have to copy it into RAM every time you want to test the BigROM or the WholeROM.

There are various tools that can be used to get the ROM files into RAM. The ARM Developer Suite includes the AXD tool. Refer to the *Getting Started on ARM* guide for instructions. Palm OS Debugger can also be used. For instructions on loading to RAM, see *Debugging a ROM*.

Debugging the ROM

For instructions on debugging, see *Debugging a ROM*.

2

Configuring Memory

This chapter describes the memory map, with emphasis on regions memory is partitioned into. Memory is configured in two passes; the first pass creates a preliminary map, the second creates a final runtime map. The events related to configuring the map during each of these passes is discussed later in this chapter, under the section on “Virtual Memory Map”.

Memory Regions

As discussed in the *Architectural Overview*, the Palm OS® memory usage is based on different kinds of memory regions:

- Read-only region. There are two read-only regions. One corresponds to the BigROM, one to the SmallROM.
- "Safe", hardware write-protectable, read-write region of memory whose contents remain untouched between soft resets and which is usually used to store databases
- Volatile read-write region whose contents will be wiped out between soft resets. This is usually used for dynamic runtime memory (stacks, variables, …)

The minimum size of each of these heaps is shown below:

Memory Area	**Size**
ROM : SmallROM	64K
ROM : BigROM	2 - 8Mbytes
Safe RAM (Storage Heap)	6 Mbytes
Volatile RAM (Dynamic Heap)	512 Kbytes

The data types used to identify the regions of memory consist of a map that specifies the number of regions, and an array of region descriptors. A region descriptor defines the kind of the region, its

base address, and its size. Note: "NonVolatile" RAM is the other name of the "Safe" RAM.

The possible types are defined in `HALMemory.h` and are as follows:

```
enum HALMemoryTag { kROM, kVolatileRAM, kNonVolatileRAM,
kSmallROM } ;

typedef Enum8 HALMemoryType;
```

The region declaration is defined as follows:

```
typedef struct HALMemoryRegionTag{
   HALMemoryType    type;
   void*           baseAddress;
   UInt32          size;
} HALMemoryRegionType;
```

The global, `GHALMemoryRegions`, is initialized as an array of HALMemoryRegionTypes. This structure contains the following information:

Type == kROM	Denotes starting address and size of the BigROM.
Type == kVolatileRAM	Denotes the starting address and size of the Dynamic Heap.
Type == kNonVolatileRAM	Denotes the starting address and size of the Storage Heap.
Type == kSmallROM	Denotes the starting address and size of the SmallROM.

This leads to the memory map declaration:

```
typedef struct HALMemoryMapTag{
     UInt8num Regions;          // number of memory regions
     const HALMemoryRegionType* regions;  // size is
                                          // numMemoryRegions
} HALMemoryMapType;
```

The routine, `HALMemoryGetMemoryMap()`, returns this structure to the calling routine. `HALMemoryGetMemoryMap()` is called

PalmSource Confidential

during the boot cycle to get a map of all available memory regions. It is also called by the flash upgrade tool to locate the smallROM so that the flash driver can be loaded and run from RAM. Refer to the chapter on "Memory" in the *DAL Reference* for more information on this function.

Virtual Memory Layout

The layout of virtual memory in the Palm OS 5 is crucial to having a flexible RAM allocation for Palm Powered devices. The initialization of the MMU page tables is heavily influenced by the layout of the virtual address space. The layout of the virtual address space influences the maximum supportable RAM configuration. Finally, the ease with which the ROM itself can be relocated (for instance between being located in Flash and being located in RAM during development) is directly affected by the virtual memory layout. To this end, the virtual address space is specified in this section. Refer to the "Initialization" chapter in the *DAL Reference* for details on the API's described in this section.

The process by which the memory map is determined at runtime is also specified in this section. During the boot process, there are several places that need to coordinate the allocation of RAM and of the virtual address space. This section specifies the various tasks that must be performed and when those tasks should be performed in a particular DAL.

Virtual Memory Map

The Virtual Memory Map defines how various portions of the virtual address space should be laid out. The Memory Map is intended to provide a reasonable amount of flexibility in the sizes of RAM and Flash available to a Licensee while still permitting the majority of the system to be independent of the actual sizes used.

The Virtual Memory Map consists of two main levels. The first is the level that is set up at Reset time. The `Reset_A` assembler routine performs the initial configuration. The mapping is then further refined during `HwrPreRAMInit()` to form the final runtime Map.

The Reset code is responsible for configuring the ARM MMU layout. The virtual address space is initially defined at reset time. The following table shows a sample set of start and end addresses.

Figure 2.1 Initial Memory Map

RAM

ROM

Direct Mapped I/O

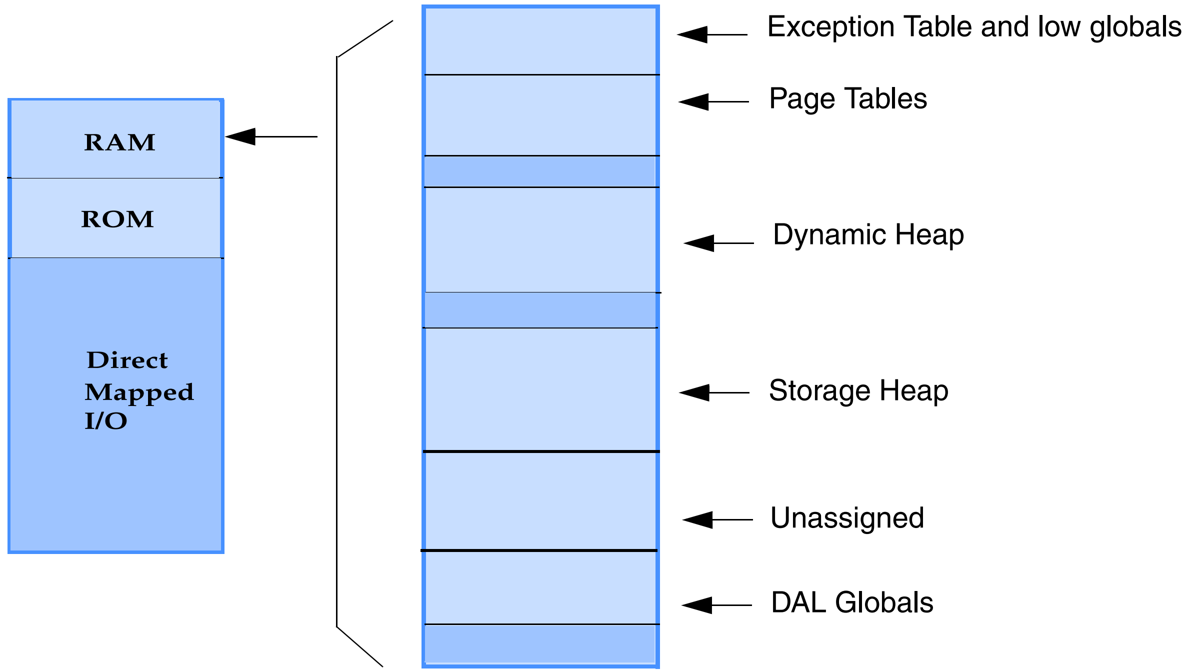
Start Address	End Address	Description
0x00000000	0x1FFFFFFF	**RAM**. This will be refined later to become the Dynamic Heap and the Storage Heap.
0x20000000	0x3FFFFFFF	**ROM**. This may be either a Masked ROM or Flash.
0x40000000	0xFFFFFFFF	**Direct Mapped I/O**. This includes On-Chip resources such as Static RAM and Hardware Registers.

It is assumed that one half gigabyte for each of RAM and ROM areas will suffice for any device running Palm OS 5. This may be an invalid assumption in later products.

After the Reset has completed, the Memory Map is further refined by `HwrPreRAMInit()` to the following RAM allocation.

PalmSource Confidential

Figure 2.2 Refined Memory Map



The following approximation on size, and sample start and end addresses can be used as a guide.

Approx. Size	Start Address	End Address	Description
16K	0x00000000	0x00003FFF	Exception Vectors and Low Memory Globals
	0x00004000	0x00007FFF	MMU Level 1 Page Table
~32K [	0x00008000	0x0001FFFF	MMU Level 2 Page Tables
~512K (*minimum*)	0x00100000	0x00100000+<A>	Dynamic Heap *
~4.5 M	0x00200000	0x00200000+<B>	Storage Heap *
~32 K (*minimum*)	0x1FF00000	0x1FF00000+<C>	DAL Static Allocation Region (DAL Globals) *

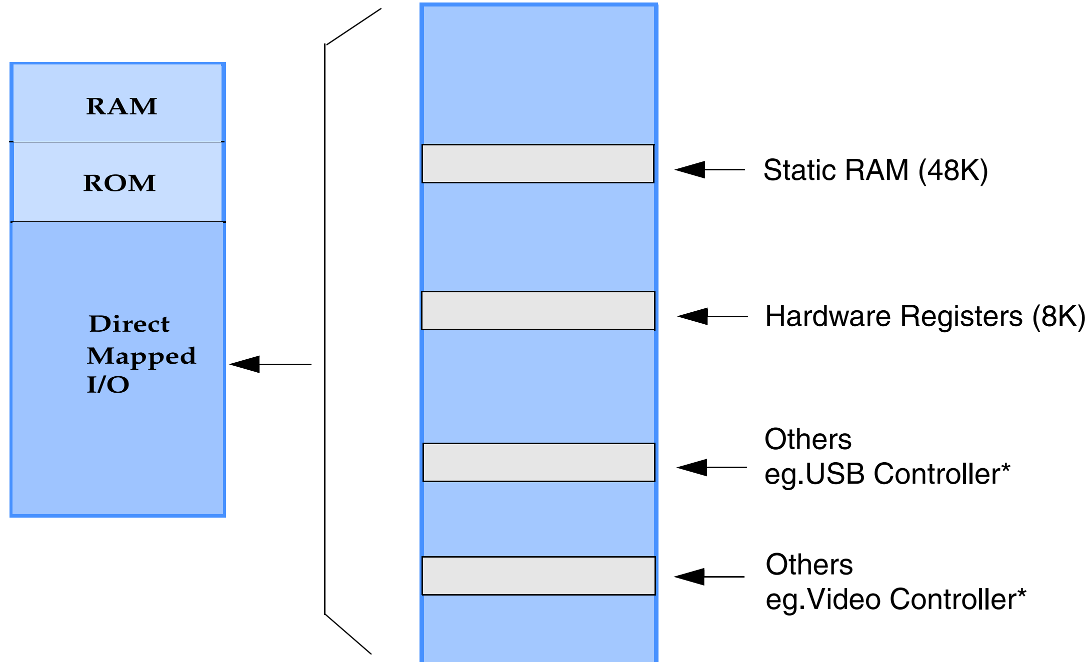
NOTE: All portions of the virtual address space that are not being used should be marked inaccessible. Also note that there ***must*** be a segment of inaccessible space immediately after the DAL Static Allocation Region. This is a requirement of the runtime environment, which exploits this space.

When configuring the memory mapping the following guidelines should be put into practice.

- *<letter>* is the size of the region. The size of these regions may end anywhere before the beginning of the next region. This is especially critical in the space immediately after the DAL Static Allocation Region.
- The table shows recommended starting addresses, however the actual starting address is determined by each licensee based on their hardware choice.
- The order of the regions must be kept as shown, however other things can be mapped in the unused space between these regions. For example, the reference DAL has a flash part used by the Smart Media driver mapped at 0x10000000 between the top of the storage heap and the beginning of the DAL Globals Region.
- The Dynamic Heap *, Storage Heap *, and DAL Static Allocation Region * must be on a 1 MB boundary. ARM architecture stipulates that there can only be one domain per MB, with a total of 16 domains in the MMU.
- There is a minimum required space for the DAL Static Allocation Region as determined by the DAL Globals structure. As of this writing a minimum of 32 kilobytes set aside for the DAL Static Allocation Region should be sufficient.

The Direct Mapped I/O area of the memory map contains the following layout.

Figure 2.3 Direct Mapped I/O Configuration



* There are no constraints on where these are located. They can go anywhere that makes sense.

HwrPreRAMInit() Responsibilities

The `HwrPreRAMInit()` function is responsible for configuring the detailed layout of memory. The refined memory map shown above (see Figure 2.2) is a good place to start. In order to establish this memory map, the MMU configuration will need to be changed to reflect the different protections of these various memory regions.

`HwrPreRAMInit()` uses `PrvMapMemoryRegion()` (see New Page Tables below) to create the final memory map based on the actual layout of the hardware and of the available RAM. The initial memory map, set up by the `Reset_A` function, serves only to get to the point where `HwrPreRAMInit()` can set up the final map. Setting up this map requires changes to many of the descriptors in the first-level table, and the creation of some second-level tables.

`HwrPreRAMInit()` needs to first determine the size of RAM. This may be determined statically at compile time, or a more

sophisticated probe can be written that will determine the physical size of RAM. Once this has been determined the amount of memory to set aside for the three main RAM areas (DAL Static Allocation region, Dynamic Heap and Storage Heap) must be determined, as well as the associated starting addresses.

The first page of RAM contains the exception tables and locations to store the RAM configuration information. This information is used by `HALGetMemoryMap()` and other routines that need to know the memory configuration, in order to ensure that they all agree on the layout of memory.

Second-level page tables need to be constructed to reflect this organization and the primary MMU page tables updated to reflect these changes.

New Page Tables

The MMU Page tables consist of a base table, officially known in ARM architecture as a "First-level table". This table contains entries called "section descriptors". Each descriptor describes one megabyte (1024 * 1024 bytes) of the virtual address space. A particular descriptor may define that megabyte of the address space, or it may reference a "Second-level table" that sub-divides the megabyte into pages, each of which is separately mapped. Getting this right is a painful process as the table is created manually. As a convenience for DAL implementors an internal function, `PrvMapMemoryRegion()`, is provided. This function provides a single entry point that takes care of this by creating new second-level page tables as necessary. This function is not exported from the DAL and exists solely for the purpose of refining the MMU settings in `HwrPreRAMInit()`. Refer to the “Initialization” chapter in the *DAL Reference* for detailed information on this routine.

The sample DAL assumes small page tables but you can create whatever kind of page tables make sense for you virtual memory layout strategy.

Low Memory Globals

The following are internal globals used by `HALGetMemoryMap()`. These globals retain the address information needed for configuring the MMU page tables during `HwrPreRAMInit()`. These globals are

defined in `Hardware73xx.h` and `Hardware73xx_A.h` and initialized in `HwrPreRamInit.c`. `Hardware73xx.h` contains data structures which define the beginning of memory, including the exception vectors, and the low memory globals described below. These globals fall immediately after the exception vectors in RAM (see Figure 2.2). When creating your own `Hardwarexxxx.h` file keep in mind that you must provide something equivalent.

`pageTableP`	Points to the next second level page table entry.
`dynamicHeapBase`	The starting address of the dynamic heap region.
`dynamicHeapSize`	The size of the dynamic heap region.
`romPhysicalBase`	The physical address of the ROM. Useful for mapping the ROM address space from RAM.
`ramSize`	The total available RAM after removing ROM space, if the ROM is in RAM.
`storageHeapSize`	The size of the storage heap region.
`flashDeviceBase`	The base address of the ROM image. This will vary depending on which ROM you are running — bigROM or smallROM.
dbgGlobals	Used by the SmallROM debugger for flashing.
dbgLockout	Set to indicate that a password has been set on the device to prevent flashing without a hard reset.

For a complete list of DAL Globals refer to Chapter 3, “DAL Globals.”

Scaling Heap Space

Licensees can control the size of the dynamic heap. The rule of thumb is based on the screen size, which is the largest consumer of

the dynamic heap. The following examples can be used in determining the configuration.

Resolution	Screen	Minimum Heap Space
High Density (1 screen buffer in memory)	320 X 320 X 16bits = 204 Kbyte	1MB
Low Density (1 screen buffer in memory)	160 X 160 X 16bits = 51 Kbyte	512 kb (default)

The smallest RAM size required by Palm OS is 4MB. RAM size can also not exceed 128MB. This includes storage and dynamic heap space combined. For the most part, applications should not assume that there is a lot of dynamic, or volatile, heap space available, other than for screen buffering.

3

DAL Globals

There are three categories of globals supported by the Palm OS® – application globals, system globals, and DAL Globals. Application globals are defined and maintained by the application developer. System globals are defined in the file Globals.h and reserved for use by the Palm OS Managers. DAL Globals are defined in several files:

DALGlobals.h	Defines container structure for DAL globals. This structure includes the HAL globals, blitter globals, screen manager globals, and kernel globals. In addition, runtime mutex identification and interrupt information is tracked by this structure. Refer to the section below on "DAL Globals Data Structure,".
HALGlobals.h	Defines HAL globals. Refer to the section below on "HAL Globals,".
HALDrawing.h	Defines blitter globals. Refer to the section below on "Blitter Globals".
HALScreenMgr.h	Defines screen manager globals. Refer to the section below on "Screen Manager Globals".

These variables are initialized and used when creating a custom DAL.

Prior to Palm OS 5 the DAL Globals were implemented as Low Memory Globals. They existed as a mapping of data structures onto the end of memory. In Palm OS 5 true global variables are supported. These variables become available once the runtime has been initialized. The DAL Globals Region, or DAL Static Allocation Region, in the Memory Map is where globals reside at runtime. Refer to "Virtual Memory Map" in Chapter 2, "Configuring Memory," for information on the mapping. Note that there are still a small number of Low Memory Globals that exist in low memory.

For instance the globals that retain the address information needed for configuring the MMU page tables during `HwrPreRAMInit()` exist as Low Memory Globals.

DAL Globals Data Structure

The DAL Globals are stored in the structure `DALGlobalsTypeStruct`.

```
typedef struct DALGlobalsTypeStruct {
   struct BltGlobalsType *bltGlobals;// see HALDrawing.h
   struct ScrGlobalsType *scrGlobals;// see HALScreenMgr.h
   void   **sysStaticBaseTable; // System Static Base Table
   Int32  ralMutexID;           // mutex ID used by runtime
   struct HALGlobalsType *HALGlobals;// ptr to the old HAL
                                     // globals
#if CPU_TYPE == CPU_ARM
   struct KernelGlobals *krnGlobals;
   struct InterruptInfoStruct **interruptInfo;
#endif
} DALGlobalsType, *DALGlobalsPtr;
```

External access to these globals is provided through the following declaration.

```
   #ifdef __cplusplus
     extern "C"DALGlobalsTypegDALGlobals;
   #else
     externDALGlobalsTypegDALGlobals;
   #endif
```

HAL Globals

Many of the HAL Globals are initialized in `LBC_ROMHardware.c`. Several globals described below must be set by each licensee.

The HAL Globals are stored in the data structure `HALGlobalsTypeStruct.` This structure is used in the definition of the `DALGlobalsTypeStruct`. The fields of

`HALGlobalsTypeStruct` are mapped to the variables listed in the table below.

Global	Values	Description
`GSysKernelDataP`	`void`	Pointer to kernel data.
`GKernelTimerCallbackP`	`void`	Kernel function to call every system "tick" so kernel can implement timeouts.
`GPenGlobalsP`	Pointer initialized to `PenGlobalsType` typedef in `PenMgrxxxxPrv.h`.	Pen manager globals.
`GKeyGlobalsP`	Pointer initialized to `KeyGlobalsType` typedef in `KeyMgrxxxxPrv.h`	Key manager globals.
`GTimGlobalsP`	Pointer initialized to `TimGlobalsType` typedef in `TimPrv.h.`	Time Manager globals.
`GSndOffTime`	`UInt32`	Number of system time units until disabling the sound.
`GSysClockFreq`	`UInt32`	Processor clock frequency.
`GHwrCPUDutyCycle`	`UInt8`	Desired Duty cycle of CPU in 31ths.

`GHwrPenDown`	`UInt8 where:`
	`0 = up`
	`<> 0 = down where number is milliseconds until next sample`
		Indicates last state of pen-down.

Global	Values	Description
GHwrCurTime	UInt32	Current hardware timer value in milliseconds. Must be non-zero at boot time to avoid keyboard repeat problems.
GHwrTotalRAMSize	UInt32	Total RAM Size.
GHwrDozeTime	UInt32	Number of milliseconds dozed since reset.
GHwrWakeUp	UInt16	Bit mask of which peripherals need to be enabled when system wakes.
GSysAutoOffSeconds	UInt16	Auto-sleep timeout in seconds.
GSysAutoOffEvtTime	UInt32	GHwrCurTime of last event. Used to support auto-off.
GSysTimerID	typedef Int32 KernelID	ID of timer used for periodic system checks, such as battery level and auto-off. Internal use only.
GSysLowMemChecksum	UInt32	Checksum of exception vector area of low memory. Used when ROM built for full error checking to make sure no applications overwrite low memory by using null pointers.
GHwrHardwareRev	UInt16	Hardware rev. This is determined during boot time. The first shipping rev is 1.
GHwrMiscFlags	UInt16	Miscellaneous hardware flags. See the section below, “Identifying Hardware Features”, for information on setting this global.

Global	Values	Description
`GHwrMiscFlagsExt`	`UInt32`	More miscellaneous hardware flags. See the section below, "Identifying Hardware Features", for information on setting this global.
`GIrq1GlobalsP`	`void`	IRQ1 handler globals.
`GIrq2GlobalsP`	`void`	IRQ2 handler globals.
`GIrq3GlobalsP`	`void`	IRQ3 handler globals.
`GIrq6GlobalsP`	`void`	IRQ6 handler globals
`GSer681GlobalsP`	`void`	Reflects value in `GIrq3GlobalsP` global.
`GSysInShutdown`	`boolean`	If true, the system is in lockout. Used for battery support.
`GSysMinWakeupThreshold`	`UInt8`	System can wake up once battery is over this voltage.
`GHwrBatteryLevel`	`UInt8`	Current battery level.
`GSysBatteryDataP`	`void`	Pointer to current battery data.
`GSysBatteryValidKindsP`	`void`	Pointer to list of valid batteries.

`GSysBatteryKind`	`UInt8 where:`
	`0 = Alkaline`
	`1 = Nickle cad`
	`2 = Lithium Ion`
	`3 = Rechargeable Alkaline`
	`4 = Nickle MH`
	`5 = Lithium Ion 1400`
	`0xFF = Other`
		SysBatteryKind enumerated type.

`GHwrBatteryGlobalsP`	`void`	Used to hold battery charging state, etc.
`GHwrBatteryPercent`	`UInt8`	Reflects battery percent.

Global	Values	Description
GSysBatteryCheckPeriod	Int16	Reflects the frequency at which the battery is checked.
GSysNextBatteryAlertTimer	Int32	Number of milliseconds until next battery warning. **NOTE:** this global is **obsolete**.
GSysBatteryCheckTimer	Int32	Number of milliseconds left before we check battery again.
GSysBatteryWarnThreshold	UInt8	If battery below this level, post low battery keyboard event.
GSysBatteryMinThreshold	UInt8	System will shut down if battery falls below this number.
GHwrStepsPerVolt	UInt16	Used to measure battery scaling for 3.x volt batteries.
GUart328GlobalsP	void	For the Serial Manager in Palm OS 3.2 and beyond, the 328 serial plugin needs its own global space since there is no global for the IRQ4 line.
GSysDayCounter	UInt16	Software day counter for the time-based critical warnings.
GHALMemoryRegions	HALMemoryRegions[] where: 0 = kROM 1 = kVolatileRAM 2 = kNonVolatileRAM 3 = kSmallROM	Global array used to store the base address and size of the four kinds of memory regions — ROM, Storage Heap, Dynamic Heap, and SmallROM. This global is initialized in `HALStaticMemInit()` in `LBC_ROMHardware.c.` For more information see Chapter 2, "Configuring Memory," in this manual, as well as the

Global	Values	Description
		chapter on "Memory" in the *DAL Reference*.
`GHALMemoryMap`	`HALMemoryMapType`	Used to hold the `GHALMemoryRegions` global, as well as the array index for this global. For more information refer to the chapter on "Memory" in the *DAL Reference*.
`GHALHardResetRequest`	`boolean`	Indicates whether a hard reset was requested by last call to `HALReset().`
`GSmallROMChecksum`	`UInt16`	Global used to identify which SmallROM booted the OS.
`GHwrOEMCompanyID`	IDs must be four ASCII characters long. Generally, alphanumerics are chosen, but this is not a requirement. For viewability, characters are limited to ASCII 33-127 (decimal). IDs consisting of all lowercase letters are reserved for use by PalmSource, Inc.	The value of this global is assigned by PalmSource Partner Engineering. This value is embedded in the ROM by each licensee, and stored in two places. In addition to the global described here, this value is stored in the Card Header of the ROM. The Card Header value is obtained from the Card ID field, which you enter before running ROM Builder. The global is set in `LBC_ROMHardware.c and` initialized during the boot cycle. When modifying the DAL and rebuilding the ROM, make sure that these two values are the

Global	Values	Description
NOTE: The company ID 'palm' is reserved for PalmSource, Inc., the maker of Palm OS while, 'Palm' is assigned to Palm, Inc., a maker of devices that run Palm OS.		same. It is suggested that you retrieve the value from the Card Header rather than hardcoding the value into the source. See the section below, "Setting Globals from the Card Header", for implementation details.
`GHwrOEMDeviceID`	`IDs must be four ASCII characters long. Generally, alphanumerics are chosen, but this is not a requirement. For viewability, characters are limited to ASCII 33-127 (decimal). IDs consisting of all lowercase letters are reserved for use by PalmSource, Inc.`	This value must be set by each by each licensee. This value need only be unique within all devices made by a particular manufacturer with the same DAL — that is, one having the same HAL ID and company ID. It is the responsibility of the DAL to distinguish between the different devices that it supports. It is the licensees responsibility to set the correct (unique) device ID in `GHwrOEMDeviceID`.

Global	Values	Description
`GHwrOEMHALID`	`IDs must be four ASCII characters long. Generally, alphanumerics are chosen, but this is not a requirement. For viewability, characters are limited to ASCII 33-127 (decimal). IDs consisting of all lowercase letters are reserved for use by PalmSource, Inc.`	This value must be set by each by each licensee. This value should be unique per DAL, within your company id. This value is embedded in the ROM by each licensee, and stored in two places. In addition to the global described here, this value is stored in the Card Header of the ROM. The Card Header value is obtained from the HAL ID field, which you enter before running ROM Builder. The global is set in `LBC_ROMHardware.c` and initialized during the boot cycle. When modifying the DAL and rebuilding the ROM, make sure that these two values are the same. It is suggested that you retrieve the value from the Card Header rather than hardcoding the value into the source. See the section below,"Setting Globals from the Card Header", for implementation details.

Global	Values	Description
GSysDispatchTableRev	UInt8	Incremented every time a trap is patched. Used by host debugger to invalidate it's cache. Note that this table can grow with every new version. If it does, the value of `HAL_GLOBALS_REGION_SIZE` will need to grow as well, in order to have enough static memory to allocate some initial stacks.
GDispExpSrcP	UInt16	Blitter buffer.
GDispRowPatBufP	UInt16	Blitter buffer.
GDispScanLine1	UInt16	Blitter buffer.
GDispScanLine2	UInt16	Blitter buffer.
GHALFreeStaticMemP	UInt8	The memory immediately after the `HALGlobalsType` and before the end of `HAL_GLOBALS_REGION_SIZE` is used to allocate static memory of various hardware managers. This is a pointer to the end of the free region.
GInitStage	UInt32	Indicates where the system is in the boot process. Modified by `HALSetInitStage()`.

Setting Globals from the Card Header

During the boot process licensees have the opportunity to identify the features of the hardware. They first have to identify the hardware before they can identify the features of the hardware. To identify the hardware, there are three globals available — Company ID, HAL ID, and Device ID. Setting these globals is described here.

The sample code below demonstrates an approved method of finding the BigROM, then subsequently calculating the card header address, verifying it really is what we think it is, and updating the Company/HAL/Device ID globals accordingly. The search loop is necessary because there are four basic memory regions (BigROM, dynamic heap, storage heap, and SmallROM) with no guarantee as to the order of each region in the map.

NOTE: In future releases there may be more than one kROM region. If this happens the code below may need to be re-written, as you may be accessing the wrong card header. In Palm OS 5 it is safe to assume the kROM region will always refer to the BigROM.

The routine below should be added to the routine `HwrPreDebugInit()` in `LBC_ROMHardware.c`. It is intended to be executed after the device's memory map is set up so that `HALMemoryGetMemoryMap()` can safely be used to find the BigROM. Simply replace the three lines that set the DAL globals with a call to this routine.

```
#ifndef BUILD_OPTION_SMALL_ROM
static void PrvInitializeDeviceIDs (void);

static void PrvInitializeDeviceIDs (void)
{
  HALMemoryMapType *    memoryMapP = HALMemoryGetMemoryMap();
  HALMemoryRegionType * romRegionP;
  CardHeaderPtr         cardP;
  UInt32                i;

  for (i = 0; i < memoryMapP->numRegions; i++)
  {
    if (memoryMapP->regions[i].type == kROM)
    {
      romRegionP = &memoryMapP->regions[i];

      if (romRegionP)
      {
        cardP = (CardHeaderPtr)(romRegionP->baseAddress);

        if (cardP && (cardP->signature == sysCardSignature))
```

```
        {
          GHwrOEMCompanyID = cardP->companyID;
          GHwrOEMHALID = cardP->halID;
          GHwrOEMDeviceID = hwrOEMDeviceIDUnspecified;  // CUSTOMIZE THIS
          return;
        }
      }
    }
  }

  GHwrOEMCompanyID  = hwrOEMCompanyIDPalmPlatform;
  GHwrOEMHALID      = hwrOEMHALIDUnspecified;
  GHwrOEMDeviceID   = hwrOEMDeviceIDUnspecified;
}
#endif
```

In layman's terms, here is what's going on.

- When booting the BigROM (but not the SmallROM), using the HAL memory regions map, find the address of the BigROM.
- If we found the BigROM, grab the Company and HAL IDs from the BigROM header.
- If anything goes wrong, give the IDs benign default values.

For detailed information on the card header layout, refer to the description on ROM Image Format in the "About Palm OS ROM Images" chapter of the *Building a ROM* manual.

Identifying Hardware Features

Once the hardware is identified, two other globals are available to tell the OS what features are present. These globals, `GHwrMiscFlags` and `GHwrMiscFlagExt`, are set with bits masks. The bit masks for these globals are defined in `HwrMiscFlags.h`. The globals themselves are initialized in the routine `HwrIdentifyFeatures()` in `ROMHiHardwareXXXX.c`.

GHwrMiscFlags Bit Masks

`hwrMiscFlagHasBacklight`	Bit to set if backlight is present.
`hwrMiscFlagHasMbdIrDA`	Bit to set if IrDA is present on the main board.
`hwrMiscFlagHasCardIrDA`	Bit to set if IrDA is present on the memory card.
`hwrMiscFlagHasBurrBrown`	Not applicable. Do not use.
`hwrMiscFlagHasJerryHW`	Not applicable. Do not use.
`hwrMiscFlagNoRTCBug`	Not applicable. Do not use.
`hwrMiscFlagHas3vRef`	Not applicable. Do not use.
`hwrMiscFlagHasAntennaSw`	Not applicable. Do not use.
`hwrMiscFlagHasCradleDetect`	Bit to set if there is an A/D converter on the HotSync port used for identifying the attached device.
`hwrMiscFlagHasSWContrast`	Bit to set if the UI should support software contrast.
`hwrMiscFlagInvertLCDForBL`	Bit to set if there is a need to invert LCD with backlight.
`hwrMiscFlagHasMiscFlagExt`	Bit to set if using `GHwrMiscFlagsExt` global.

GHwrMiscFlagsExt Bit Masks

`hwrMiscFlagExtSubIDMask`	Bit to set for subtype ID Mask. 'AND' this bit to `GHwrMiscFlagsExt` after setting all other bits.
`hwrMiscFlagExtHasLiIon`	Bit to set if there is a Lithium Ion battery that is rechargable in the cradle.
`hwrMiscFlagExtHasRailIO`	Not applicable. Do not use.

`hwrMiscFlagExtHasFlash`	Bit to set if there is Flash ROM.
`hwrMiscFlagExtHasFParms`	Bit to set if there is a Flash parameters area.
`hwrMiscFlagExt115KIrOK`	Bit to set if the device supports 115K IR transfers.
`hwrMiscFlagExtHasExtLCD`	Not applicable. Do not use.
`hwrMiscFlagExtHasSWBright`	Bit to set if the device has software controlled brightness.
`hwrMiscFlagExtNeedsLpr`	Not applicable. Do not use.

Blitter Globals

The blitter globals are stored in the structure `BltGlobalsType`. This structure is a field in the `DALGlobalsTypeStruct`. It is accessed through the variable `GBltGlobals`. The routine `HALDrawInit()` initializes the blitter globals.

```
typedef struct BltGlobalsType {
   UInt32 userColorMask;
   void * BitonalBlitCopyFuncs[5];
   void * BitonalBlitOverFuncs[5];
   UInt8 translateSpace[256];      // used as temp translation table & for
                                   // winPaintInverse translation
   UInt16 translate16Space[256];   // maps indexed pixels into 16-bit pixels
                                   // for 16-bit output

   BltBitmapType canvasBitmap;

   ExpandedPatternInfo patInfoStorage;
} BltGlobalsType;

#define GBltGlobals(*(BltGlobalsType*)DALGlobalsP->bltGlobals)
```

Screen Manager Globals

The screen manager globals are stored in the structure `ScrGlobalsType`. This structure is a field in the `DALGlobalsTypeStruct`. It is accessed through the variable `GScrGlobals`.

```
// Screen Manager globals
typedef struct ScrGlobalsType
   {
   // The fields in this section mirror the bitmap and auxiliary structures
   // (color table, directInfo structure used by 16 bit bitmaps, and the
   // pointer to the bitmap data).  The runtime screen data structures probably
   // do not map to these fields, since the fields correspond to structures
   // used by a 16 bit screen.  A 4 bit screen, for example, will only have
   // 16 color table entries, and will not have a directInfo structure.  When
   // the screen depth changes, baseAddr is repositioned according to the size
   // of the color table and whether the directInfo structure exists.

   BitmapType      bitmap;       // a pointer to this field is stored in the
                                 // bitmapP field of onscreen windows

   ColorTableType  colorTable;            // bitmap color table
   RGBColorType    colorEntries[256];
   BitmapDirectInfoType directInfo;       // used in direct color

   MemPtr          baseAddr;     // Following the bitmap (flags indirect)
                                 // is the address (part of bitmapType)


   MemPtr   lockedAddr;          // screen buffer used by HALScreenLock
   ScreenTransferFunc screenTransferFuncP;
   MemPtr   screenBuf;           // hardware screen buffer
   MemPtr   transferBuf;         // intermediate buffer used by
                                 // HALScreenSendUpdateArea

   AbsRectType     updateR;      // update rect for HALScreenSendUpdateArea
   UInt32          lastUpdate;   // Tickcount of last update

   // Color translation tables. These map pixels values from other depths
   // with standard system cluts into a pixel value for the screen's current
   // depth and clut. They get updated by WinPallete() when it is called to
   // update the palette on the screen's bitmap.

   UInt16* colorTranslateP[4];   // 4 color translate table to current palette
```

```
                                // [0] for 1 bit depth to current,
                                // [1] for 2 bits etc...

  UInt16  screenLockCount;      // number of times HALScreenLock called
  Boolean doDrawNotify;         // call HALScreenDrawNotify after drawing
  ScrPaletteState paletteState8Bit; // state of the screen bitmap's palette
  }
ScrGlobalsType
```

4

Dynamic Input Area

This section describes how to customize the **dynamic input area** available on some Palm Powered™ handhelds. A dynamic input area is a software implementation of the input area that is traditionally silkscreened onto the device. Implementing the area as software allows the user to expand and collapse the area at will, giving more space to the display of application data when it is needed.

A control bar at the bottom of the screen contains various buttons, including one, called the trigger, that opens the input area when it is closed. A trigger also typically appears in the input area to provide a way for the user to close the input area. See Chapter 5, “Using the Control Bar,” for more information.

The control bar is typically closed when the input area is open, except on double density screens, when it is always shown; however, licensees can change this behavior by using alternative input area and control bar bitmaps.

Input Area Resources

The dynamic input area use bitmaps and related 'silk' resources that are stored in the HAL. The 'silk' resource is generated from an XRD file that contains the following definitions:

- application area dimensions, alphabetic and numeric input areas, soft buttons, and a virtual keystroke associated with each soft button for five different states:
  - input area open, without trigger
  - input area open, with trigger soft button
  - control bar open, with trigger (input area closed)
  - control bar open, without trigger (input area closed)
  - full screen application area (input area and control bar both closed)

- bitmap family definitions for the input area and control bar bitmaps

The XRD file and the bitmaps are stored in the DAL's DevResources component. The directory contains the input area bitmaps listed in Table 4.1. The resource ID constants for the bitmaps are defined in `HALScreenMgr.h`.

Table 4.1 Input area bitmaps

Bitmap Description	Resource ID Constant
Input area with no trigger	`inputAreaBitmap`
Input area with no trigger, selected mode	`inputAreaSelectedBitmap`
Input area with trigger	`inputAreaBitmapTrigger`
Input area with trigger, selected mode	`inputAreaSelectedBitmapTrigger`

The OS reads the bitmap resources and sends them to the display driver when booting, and whenever the input area or control bar bitmap changes in response to the opening or closing of the input area, or a changed state of the input area trigger.

Input Area Feature Flags

The HAL is responsible for defining the dynamic input area feature flags returned by the `sysFtrNumInputAreaFlags` selector of `FtrGet`. These flags identify what input area features the device supports; they are defined in `Graffiti.h`:

```
#define grfFtrInputAreaFlagDynamic 0x00000001
#define grfFtrInputAreaFlagLiveInk 0x00000002
#define grfFtrInputAreaFlagCollapsible 0x00000004
```

The flags must be defined using `FtrSet` in `HALSetInitStage` when the `uiValue` argument is `kWinMgrOKStage`.

For example, if the device supports a dynamic input area, live ink, and a closable (collapsible) input area, the following logic should be added to `HALSetInitStage`:

```
if (uiValue == kWinMgrOKStage)
                  {
   Err err;
   RectangleType bounds;

   // check to see if display driver supports dynamic input area
   err = HALDisplayGetAttributes(kHALDispInputAreaLoc, (UInt32*) &bounds);
   if (err == errNone)
   {
   // if dynamic input area supported, set all of the input area flags
   FtrSet(sysFtrCreator, sysFtrNumInputAreaFlags, grfFtrInputAreaFlagDynamic |
      grfFtrInputAreaFlagLiveInk | grfFtrInputAreaFlagCollapsible);
   }
}
```

5

Using the Control Bar

This section describes how to customize the **control bar** available on some Palm Powered™ handhelds. A control bar is displayed across the bottom of the screen, below the input area. It is used to hold shortcut icons to specific applications. Unlike a status bar, a control bar does not contain icons to applications that monitor state. Control bar icons offer an alternative to starting an application through the launcher.

The control bar contains various buttons, including one, called the trigger, that opens the input area when it is closed. A trigger also typically appears in the input area to provide a way for the user to close the input area. See Chapter 4, "Dynamic Input Area," for more information.

The control bar is typically closed when the input area is open, except on double density screens, when it is always shown; however, licensees can change this behavior by using alternative input area and control bar bitmaps.

Control Bar Resources

As with the dynamic input area, the control bar uses bitmaps and related 'silk' resources that are stored in the HAL. The 'silk' resource is generated from an XRD file that contains the following definitions:

- application area dimensions, alphabetic and numeric input areas, soft buttons, and a virtual keystroke associated with each soft button for five different states:
  - input area open, without trigger
  - input area open, with trigger soft button
  - control bar open, with trigger (input area closed)

  - control bar open, without trigger (input area closed)
  - full screen application area (input area and control bar both closed)
- bitmap family definitions for the input area and control bar bitmaps

The XRD file and the bitmaps are stored in the DAL's DevResources component. The directory contains the control bar bitmaps listed in Table 5.1. The resource ID constants for the bitmaps are defined in `HALScreenMgr.h`.

Table 5.1 Control bar bitmaps

Bitmap Description	Resource ID Constant
Control bar with trigger	`controlBarBitmap`
Control bar with trigger, selected mode	`controlBarSelectedBitmap`
Control bar with no trigger	`controlBarNoTriggerBitmap`
Control bar with no trigger, selected mode	`controlBarNoTriggerSelectedBitmap`

The OS reads the bitmap resources and sends them to the display driver when booting, and whenever the input area or control bar bitmap changes in response to the opening or closing of the input area, or a changed state of the input area trigger.

Customizing the Control Bar

The control bar can be customized to display additional buttons. You can also remove buttons, such the Bluetooth, if this feature is not part of your offering.

Editing the Bitmaps

Before you begin you should be aware of how the control bar bitmaps are paired up. The reference bitmaps are located in `Platform/DAL/Common/SampleDevResources/Rsc/bitmaps` directory.

Each control bar bitmap works as a set, with one black bitmap and one non-black (grey or blue). The non-black bitmap is the "selected mode" version and is used to show a change in color when a button is selected. The following example shows how the two images are used together. The top image, `Bitmap_19025QX-8.bmp`, is the default bitmap. The bottom image, `Bitmap_19125QX-8.bmp`, is the bitmap used when a button is selected. The coordinates of the button are used to determine which section of the selected bitmap to display over top of the default bitmap so that only the button changes color when selected.

Figure 5.1 Default and Selected Control Bar

The other pairing of bitmaps is has to do with the trigger. If you intend to provide a collapsible input area, you must use bitmaps with the trigger. However, because there are scenarios where the input area must remain collapsed, and the control bar is the only thing displayed across the bottom of the screen, you need to have the control bar without the trigger available. Because of this you will not only need to edit the default/selected bitmaps, you will also need to modify the set of bitmaps that contain the trigger. So, in addition to the two example images above, you would also need to modify the two bitmaps in Figure 5.2, `Bitmap_19020QX-8.bmp` and `Bitmap_19120QX-8.bmp`.

Figure 5.2 Default and Selected with Trigger

Adding Buttons

The following steps guide you through the process of adding buttons to the control bar. Note that these instructions use the reference code in their descriptions. It is expected that you will have a copy of the files in your own development environment.

1. Add the icon for the button to the appropriate bitmap files.
2. Define a new virtual character for the button. There are pre-defined ranges assigned to each licensee. Search for your range in the `chars.h` file located in `Platform/Incs/Core/Common`. The new vchar should be look something like this:

   `#define vchr`*<app/operation> <number in range>*

   If you want to give developers access to this vchar, you need to expose this definition in a public header file. Otherwise you can add it to a private header file.
3. Add the button coordinates and definition to the appropriate InputArea XRD file.The reference XRD files are located in the `Platform/DAL/Common/SampleDevResources/Rsc` directory. There are several flavors of the InputArea XRD files as follows:
   - InputArea-1X.xrd. (Single Density Display)
   - InputArea-2X.xrd (Double Density Display)
   - InputArea-QX.xrd (QVGA Display)

– InputAreaAllDensities.xrd (Simulator use only. Single, Double, and QVGA Display)

Open the InputArea file and locate the `<SILK_SCREEN_RESOURCE>` for the control bars you will be using. The `COMMENT` field will tell you which control bar the block of definitions applies to. You must modify ***two*** control bar definitions; one with a trigger and the matching one without. These definitions define the coordinates for both the default and the selected display area, so you only need to account for the control bars with and without the trigger. Add the following button definition before the end of the block:

```
<SILK_SCREEN_BUTTON>
   <BOUNDS>
      <LEFT> left coordinate </LEFT>
      <TOP> top coordinate </TOP>
      <WIDTH> width </WIDTH>
      <HEIGHT> height </HEIGHT>
   </BOUNDS>
   <KEY_CHAR> vchar code </KEY_CHAR>
   <KEY_CODE> 0x0000 </KEY_CODE>
   <KEY_MODIFIERS> 0x0008 </KEY_MODIFIERS>
</SILK_SCREEN_BUTTON>
```

where:

left coordinate	=	rounded down (left coordinate/resolution)
top coordinate	=	rounded down (top coordinate/resolution)
width	=	rounded up(width/resolution)

height	=	rounded up(height/resolution) Note that the original coordinates (left, right, width, and height) must be translated so that they are using a width of 160, even if you are using a higher resolution screen. The number to use for resolution in the above formulas are as follows: • 1 for 1X • 1.5 for QVGA • 2.0 HVGA
vchar code		New virtual character code
`<KEY_CODE>`		Leave as is. This field is currently unused.
`<KEY_MODIFIERS>`		Must be `0x0008`

4. Finally, you must patch `SysHandleEvent()` to do a `SysUIAppSwitch()` to the application associated with the button. Refer to the *Shared Library Design Guide* for information on writing a patch.

Removing Buttons

To remove a button from the control bar simply follow the instructions above in reverse.

1. If you have installed a patch to `SysHandleEvent()` to do a `SysUIAppSwitch()` to the application associated with the button, remove the patch. **Note:** if you are removing one of the buttons provided in the reference bitmaps, for instance Bluetooth, this step is not necessary.
2. Remove the button coordinates and definition from the appropriate InputArea XRD file.
3. Remove the virtual character definition for this button from the appropriate header file.
4. Remove the icon from all appropriate control bar bitmap files.

6

Replacing System Fonts

This chapter describes how to replace the Palm OS® system fonts with your own fonts. There are eight Palm OS system fonts, as listed in Table 6.1. The font IDs shown in Table 6.1 are used to access these fonts. (A **font ID** is an index into the system's font list.)

Table 6.1 Palm OS built-in fonts

Index	Font ID	Description
0	`stdFont`	A small standard font used to display user input. This font is small to display as much text as possible.
1	`boldFont`	Same size as `stdFont` but bold for easier reading. Used for text labels in the user interface.
2	`largeFont`	A larger font provided as an alternative for users who find the standard font too small to read. This font is used as the default on single-density Japanese systems.
3	`symbolFont`	Contains many special characters such as arrows, Graffiti 2® Shift Indicators, and so on.
4	`symbol11Font`	Contains the check boxes, the large left arrow, and the large right arrow.

Table 6.1 Palm OS built-in fonts *(continued)*

Index	Font ID	Description
5	`symbol7Font`	Contains the up and down arrows used for the repeating button scroll arrows and the dimmed version of the same arrows.
6	`ledFont`	Contains the numbers 0 through 9, –, ., and the comma (,). Used by the Calculator application for its numeric display.
7	`largeBoldFont`	Same size as `largeFont` but bold.

The fonts that Palm OS uses by default are included in two different places. The `System.prc` file contains the **symbol fonts** (`symbolFont`, `symbol11Font`, `symbol7Font`, and `ledFont`). The locale module provides the rest of the fonts (**text fonts**, or fonts used for text).

You can replace the system fonts with your own fonts in one of the following ways:

- Create a font database and include it in the ROM. If this database exists, the fonts it contains override any fonts already installed. See "Creating a Font Database" on page 47.
- Replace the entire locale module in the ROM as described in "Replacing Fonts in the Locale Module" on page 48.

The first method of replacing system fonts is recommended in these instances:

- You want to replace only the symbol fonts.
- You want to replace the text fonts in a ROM that uses the Palm Latin character encoding.

If the ROM uses the Palm Shift JIS or any other Asian encoding, it is better to replace the entire locale module if you want to replace the fonts used for text. Replacing the locale module is less error prone in this instance because of the number of resources involved. It also saves memory because fonts for double-byte character sets are quite large.

Note that creating a font database is only supported in Palm OS 5 release 5.1 or later. If you are using an earlier version of the OS, you must replace the system fonts by replacing the locale module.

Creating a Font Database

To override the system fonts using a font database, do the following:

1. Create the resources for the fonts you want to replace. See "Creating Font Resources" on page 59.

   You can replace all of the system fonts or a subset of them.

   The resources you create ***must*** have the same resource type and resource ID as the ones you are replacing. Palm Latin ROMs use the font resource types and IDs listed in Table 6.2. ROMs compiled for ARM processors use the ARM resource type. Palm Shift JIS ROMs use the same font resources for the four symbol fonts but have different resources for the text fonts.
2. Compile these font resources into a PRC database with the type `'fnts'` and the creator `'psys'` for the four standard fonts.
3. Include the database in the ROM that you build.

Table 6.2 Palm Latin built-in fonts

Font ID	ARM Resource Type	Resource ID
`stdFont`	`'afnx'`	32000
`boldFont`	`'afnx'`	32001
`largeFont`	`'afnx'`	32002
`symbolFont`	`'afnx'`	10000
`symbol11Font`	`'afnx'`	10001
`symbol7Font`	`'afnx'`	10002
`ledFont`	`'afnx'`	10003
`largeBoldFont`	`'afnx'`	32003

PalmSource Confidential

When Palm OS initializes, it loads the locale module and then it looks for a database with type `'fnts'` and creator `'psys'`. If that database exists, it uses any fonts in that database in place of the default system fonts.

Replacing Fonts in the Locale Module

PalmSource provides one locale module for each supported character encoding, and each ROM uses only one locale module. The `charEncodingPalm`*XXX* values are defined in `PalmLocale.h`, which is distributed with the standard headers. The locale modules associated with these encoding values include `ShiftJISLocModule.prc` (Japanese), `LatinLocModule.prc` (Latin), and `GBLocModule.prc` (Simplified Chinese).

To replace the fonts in the locale module, do the following:

1. Decompile the locale module into an XRD file using the **GenerateXRD** tool.
2. Edit the font resources in the file.
3. Recompile the locale module into a PRC file using **PalmRC**.
4. Build the ROM. The locale module PRC file should already be a part of the ROM project.

The next two sections show example commands for decompiling and recompiling the locale module PRC. You can find more information about these tools in the documents *Building Palm OS Application Interfaces* and *Localization Guide*, both of which are included in the PDK.

GenerateXRD

Use the GenerateXRD tool to decompile the locale module into an XRD source file. Normally this tool is used to extract only user interface resources. When editing a PRC file, you must use extra options so that all resources and PRC file attributes are preserved during the editing process. For example:

```
GenerateXRD ShiftJISLocModule.prc -o ShiftJISLocModule.xrd -d
-s -target 4.0J
```

This command is composed of the following elements:

- The option `ShiftJISLocModule.prc` specifies the input file to be decompiled.
- The option `-o ShiftJISLocModule.xrd` specifies that the output file is `ShiftJISLocModule.xrd`.
- The option `-d` specifies that the PRC database header attributes should be output to the XRD file.
- The option `-s` specifies that "segment" resources such as code and overlay resources should be output to the XRD file.
- The option `-target 4.0J` tells the tool how to correctly convert from the Shift-JIS character encoding into UTF-8 (Unicode), which is the character encoding used for XRD files.

After you have generated the XRD file, edit it (using a UTF-8 compatible text editor) to include your hierarchical fonts as described in "Creating a Hierarchical Font."

PalmRC

After you are finished editing the XRD file, compile it back into a PRC file using the PalmRC tool. For example:

```
PalmRC ShiftJISLocModule.xrd -o ShiftJISLocModule.prc -p ARM
-target 4.0J
```

This command is composed of the following elements:

- The option `ShiftJISLocModule.xrd` specifies the input file to be compiled.
- The option `-o ShiftJISLocModule.prc` specifies that the output file is `ShiftJISLocModule.prc`.
- The option `-p ARM` specifies that resources should be compiled as ARM format.
- The option `-target 4.0J` preserves the resource text encoding for the Japanese resources.

Once you have the PRC file you are ready to build the ROM.

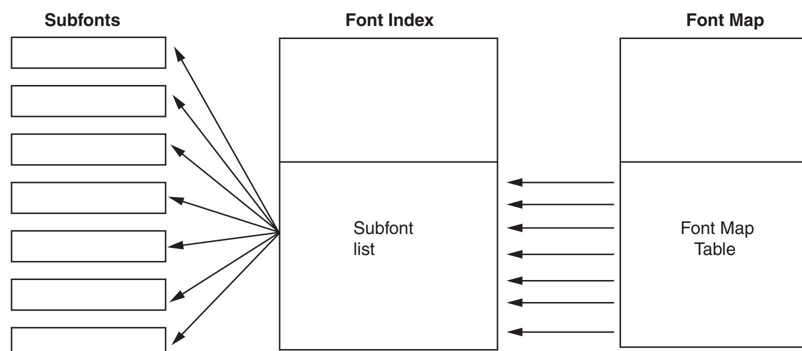
Creating a Hierarchical Font

Hierarchical fonts are fonts used to support double-byte character encodings such as those used for the Japanese, Chinese, or Korean alphabets. A hierarchical font is composed of the following resources (see Figure 6.1):

- **Subfonts**, which are the same font resources that are used for fonts in single-byte character encodings. A subfont defines at most 256 glyphs for a double-byte character encoding (which by definition have well over 256 characters). A hierarchical font contains as many subfonts as are required to provide all of the necessary glyphs for the encoding.
- One **font map** resource that tells Palm OS which subfonts to use for which character codes

The **font index resource**, which is used to build the system's internal array of font pointers, is also integral in the workings of a hierarchical font. The font map uses the list built from the font index resource when identifying subfonts.

Figure 6.1 Hierarchical font



The rest of this section describes each of these three resource types in more detail and then gives an example of how Palm OS uses these resources to determine how to draw characters on the screen.

Subfonts

A subfont defines a subset of the glyphs for a double-byte character encoding. A subfont contains either the set of single-byte characters supported by the character encoding, or all of the double-byte

glyphs that have the same high byte value. A subfont may be either a standard font resource (which supports only single-density displays) or an extended font resource (which supports any mix of single, 1.5X, or double-density data). Note however that each font ***must*** contain single density data, for compatibility with applications that draw text into single density offscreen bitmaps. These resources are the same as those used for single-byte fonts. See "Creating Font Resources" on page 59 for more information about these resources and how to create them.

You can mix and match standard fonts and extended fonts within a hierarchical font. Because extended fonts with multiple densities can be much larger than standard fonts, you might find it advantageous to use a standard font resource for infrequently used subsets of characters and use extended font resources for the frequently used characters. On high-density displays, Palm OS pixel-doubles the low-density glyphs if the double-density glyphs are not available.

When you create a resource to be used as a subfont, ensure the following:

- The font type element should have the value 0x9400 for a standard font resource or 0x9600 for an extended font resource.
- Each subfont must be as tall or taller than the height indicated by the font map metrics that are specified in the font map's header (see "Font Map Resources" on page 53). The height is defined as the ascent plus the descent.
- The Palm OS blitter does not attempt to align the baselines of subfonts, so it is best if all subfonts use the same value for the ascent.

The Font Index Resource

The font index resource identifies all of the fonts available to the system. The font index is a resource of type `'afti'` on ARM with the resource ID 32000. Palm OS uses the font index to build its internal list of system fonts and subfonts. Listing 6.1 shows the XML definition of a font index resource.

Listing 6.1 Font index resource

```
<FONT_INDEX_RESOURCE RESOURCE_ID="32000" COMMENT="ShiftJIS
Font Resources Index">

     <FONT_INDEX_ITEMS>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'aftm' </FONT_RES_TYPE>
           <FONT_RES_ID> 32000 </FONT_RES_ID>
        </FONT_INDEX_ITEM>
        <FONT_INDEX_ITEM>

           <FONT_RES_TYPE> 'aftm' </FONT_RES_TYPE>
           <FONT_RES_ID> 32001 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'aftm' </FONT_RES_TYPE>
           <FONT_RES_ID> 32002 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'afnx' </FONT_RES_TYPE>
           <FONT_RES_ID> 10000 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'afnx' </FONT_RES_TYPE>
           <FONT_RES_ID> 10001 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'afnx' </FONT_RES_TYPE>
           <FONT_RES_ID> 10002 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'afnx' </FONT_RES_TYPE>
           <FONT_RES_ID> 10003 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

        <FONT_INDEX_ITEM>
           <FONT_RES_TYPE> 'aftm' </FONT_RES_TYPE>
           <FONT_RES_ID> 32003 </FONT_RES_ID>
        </FONT_INDEX_ITEM>

     </FONT_INDEX_ITEMS>
```

```
  <FONT_INDEX_EXT_ITEMS>
    <FONT_RES_ID> 32000 </FONT_RES_ID>
    <FONT_RES_ID> 32001 </FONT_RES_ID>
    ...
  </FONT_INDEX_EXT_ITEMS>

</FONT_INDEX_RESOURCE>
```

The font index has two parts:

1. A System Font list (`FONT_INDEX_ITEMS`), which lists the resource types and IDs used for the default fonts. The default fonts are listed in Table 6.1 on page 45.

   You must provide a list with eight elements for the system fonts. The system fonts are listed in Table 6.1 on page 45. For the symbol fonts, use the resource types and IDs shown in Listing 6.1.

   For the four standard fonts, list the resource type and ID for the font map resource. You do not have to provide four different fonts, but you still should have four different entries. (For example, you might decide to have `largeFont` and `largeBoldFont` use the same font map.)

2. A Subfonts list (`FONT_INDEX_EXT_ITEMS`), which lists the resource IDs of the fonts used as subfonts within all hierarchical fonts in the system.

   The font map resource uses the index into the list to identify the subfonts that it uses.

   If there are no hierarchical fonts installed in the system, the subfonts list is empty.

Font Map Resources

A font map is a resource of type `'aftm'` on ARM that maps the character codes within a character encoding to the subfonts that contain the glyphs for those character codes. Listing 6.2 shows the XML definition of the font map resource.

PalmSource Confidential

Listing 6.2 Font map resource

```
<FONT_MAP_RESOURCE RESOURCE_ID="32000">

   <FONT_MAP_FONT_TYPE>      0xC000 </FONT_MAP_FONT_TYPE>
   <FONT_MAP_FIRST_CHAR>     0x0000 </FONT_MAP_FIRST_CHAR>
   <FONT_MAP_LAST_CHAR>      0xFFFF </FONT_MAP_LAST_CHAR>
   <FONT_MAP_MAX_WIDTH>      0x000B </FONT_MAP_MAX_WIDTH>
   <FONT_MAP_KERN_MAX>       0x0000 </FONT_MAP_KERN_MAX>
   <FONT_MAP_N_DESCENT>      0x0000 </FONT_MAP_N_DESCENT>
   <FONT_MAP_F_RECT_WIDTH>   0x000B </FONT_MAP_F_RECT_WIDTH>
   <FONT_MAP_F_RECT_HEIGHT>  0x000B </FONT_MAP_F_RECT_HEIGHT>
   <FONT_MAP_OWT_LOC>        0x0000 </FONT_MAP_OWT_LOC>
   <FONT_MAP_ASCENT>         0x0009 </FONT_MAP_ASCENT>
   <FONT_MAP_DESCENT>        0x0002 </FONT_MAP_DESCENT>
   <FONT_MAP_LEADING>        0x0000 </FONT_MAP_LEADING>
   <FONT_MAP_ROW_WORDS>      0x0000 </FONT_MAP_ROW_WORDS>

   <FONT_MAP_TABLE>

      <FONT_MAP_ENTRY>
         <FONT_MAP_ENTRY_FLAGS> 0x00 </FONT_MAP_ENTRY_FLAGS>
         <FONT_MAP_ENTRY_STATE> 0x01 </FONT_MAP_ENTRY_STATE>
         <FONT_MAP_ENTRY_VALUE> 0x00 </FONT_MAP_ENTRY_VALUE>
      </FONT_MAP_ENTRY>

      <FONT_MAP_ENTRY>
         <FONT_MAP_ENTRY_FLAGS> 0x00 </FONT_MAP_ENTRY_FLAGS>
         <FONT_MAP_ENTRY_STATE> 0x01 </FONT_MAP_ENTRY_STATE>
         <FONT_MAP_ENTRY_VALUE> 0x00 </FONT_MAP_ENTRY_VALUE>
      </FONT_MAP_ENTRY>

      ... etc. to make exactly 256 entries

   </FONT_MAP_TABLE>

</FONT_MAP_RESOURCE>
```

There are two main parts to a font map resource: a header, which provides metrics for the subfonts, and a font map table, which maps the character codes to the subfonts. Table 6.3 describes the elements that comprise the font map's header.

Table 6.3 Font map header elements

Element	Description
FONT_MAP_FONT_TYPE	A mask providing the general characteristics of the font. When creating a font map, use the value 0xC000.
FONT_MAP_FIRST_CHAR	This value is not used and must be set to 0.
FONT_MAP_LAST_CHAR	This value is not used and must be set to 0xFFFF.
FONT_MAP_MAX_WIDTH	The maximum width in pixels of any glyph. In Palm OS, there is currently no difference between this element and FONT_MAP_F_RECT_WIDTH.
FONT_MAP_KERN_MAX	This value is not currently used and must be set to 0.
FONT_MAP_N_DESCENT	This value is not currently used and must be set to 0.
FONT_MAP_F_RECT_WIDTH	A metric of the font image. In Palm OS, this metric is equivalent to the maximum width in pixels of any glyph in the font.
FONT_MAP_F_RECT_HEIGHT	The maximum height of the glyphs in this font. This value is FONT_MAP_ASCENT plus FONT_MAP_DESCENT.
FONT_MAP_OWT_LOC	This value is not used and must be set to 0.
FONT_MAP_ASCENT	The distance in pixels from the top of the font rectangle to its baseline.

PalmSource Confidential

Table 6.3 Font map header elements *(continued)*

Element	Description
FONT_MAP_DESCENT	The distance in pixels from the baseline to the bottom of the font rectangle.
FONT_MAP_LEADING	The font's leading, which is the vertical space between lines of text, in pixels. This field is unused in Palm OS and must be set to 0. If your font requires a leading value, add blank space to the bottom or top of each of your glyphs.
FONT_MAP_ROW_WORDS	This value is not used and must be set to 0.

The FONT_MAP_TABLE element is the table that maps character codes to subfonts. This table has 256 entries of type FONT_MAP_ENTRY. Each FONT_MAP_ENTRY contains the elements listed in Table 6.4.

Table 6.4 Font map table entry elements

Element	Description
FONT_MAP_ENTRY_FLAGS	Not used.
FONT_MAP_ENTRY_STATE	One of the following: fntStateIsChar (1) — The current index is a single-byte character, and it is used to determine which glyph in the specified subfont is used for this character.

Table 6.4 Font map table entry elements *(continued)*

Element	Description
	`fntStateNextIsChar` (2) — The current index is the first byte of a double-byte character. This means that the next byte is used to determine which glyph in the specified subfont is used for the character.
`FONT_MAP_ENTRY_VALUE`	The subfont that contains the glyphs for the character or the characters that begin with this byte. The subfont is identified by its index into the Subfonts list in the font index resource. For more information, see "The Font Index Resource" on page 51. This value is an absolute index. Suppose you define the four system fonts and for each of them, you use 45 separate font resources. If you list all of the subfonts in order of use, `stdFont` uses subfonts 0 through 44, `boldFont` uses subfonts 45 through 89, `largeFont` uses subfonts 90 through 134, and so on. Also note that subfonts may be reused. For example, `stdFont` and `boldFont` may use the same subfonts for a certain set of characters.

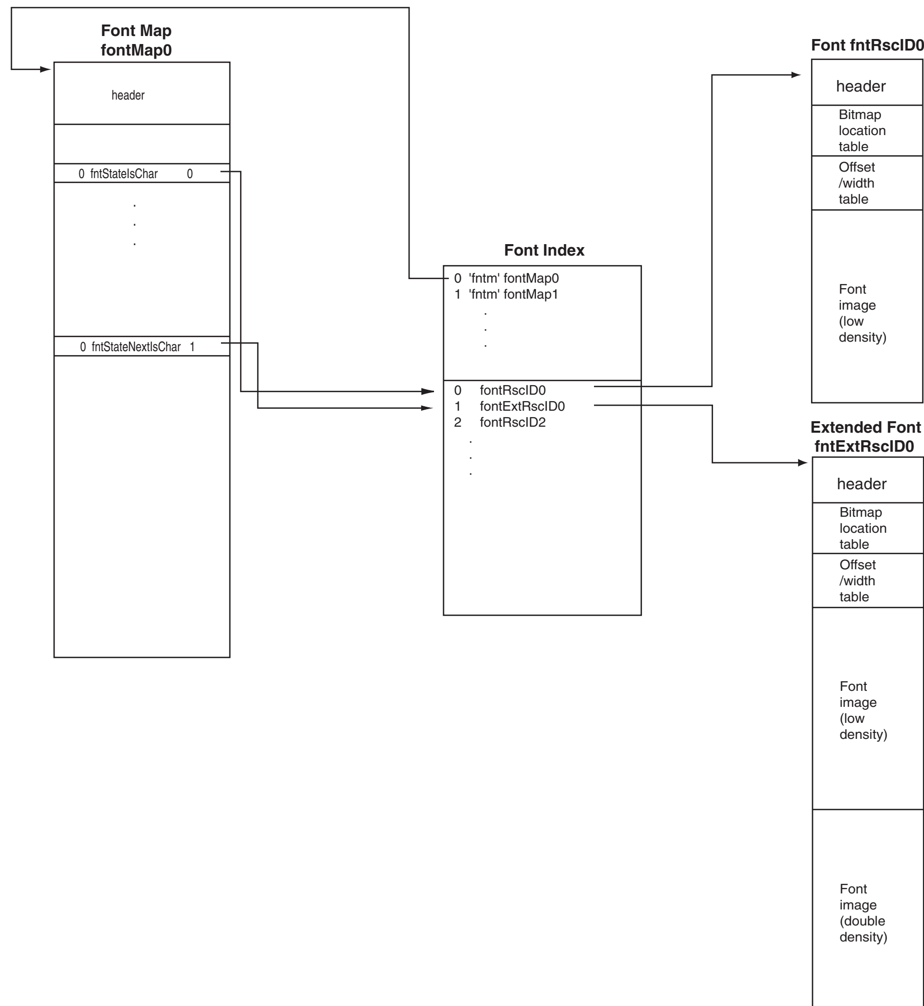
How a Hierarchical Font Works

Suppose an application contained the following code to write two characters to the display:

```
FntSetFont(stdFont); // set the font to be used
WinDrawChar(0x28, 50, 50);
WinDrawChar(0x806A, 50, 60);
```

To locate the glyphs necessary to draw these characters to the screen, Palm OS does the following (see Figure 6.2):

Figure 6.2 Hierarchical font example



1. It looks up `stdFont` (font ID 0) in the array built from the font index resource's System Fonts list to determine which resource is used as the standard font.

   As shown in Figure 6.2, the standard font is a font map with the ID `fontMap0`.

2. For the first character, Palm OS uses 0x28 as the index into the font map. It learns that 0x28 is a single-byte character (its state is `fntStateIsChar`) and its glyph is in Subfont 0.
3. Palm OS finds the pointer to Subfont 0 in the internal array of subfont pointers that it built from the font index resource at boot time.
4. It uses 0x28 as the character to draw with this subfont.
5. Next, Palm OS needs the glyph for character 0x806A. It uses the first byte, 0x80, as the index into the font map. This entry has the state `fntStateNextIsChar` and the subfont value 1. This means that Palm OS should use the next byte (0x6A) as the character to draw with this subfont.
6. Palm OS finds the pointer to Subfont 1 in the internal array of subfont pointers. This subfont is used to draw all characters that begin with 0x80.
7. Because subfont 1 is a pointer to an extended font, the current drawing state's density determines which glyph bitmap is used to draw the character.

Creating Font Resources

This section describes how to create standard font and extended font resources. You need this information both for building a system font database to replace the fonts in the Palm Latin character encoding and for creating a hierarchical font for the Palm Shift JIS character encoding within a locale module.

For information on including font resources in a font database, see "Creating a Font Database" on page 47. For information on including font resources in a hierarchical font, see "Creating a Hierarchical Font" on page 50.

Standard Font Resources

A standard font resource defines the glyphs for a single-byte character encoding (such as Palm Latin) or for up to 256 characters with the same high byte value in a double-byte character encoding. Standard font resources support a single screen density of 160 X 160 pixels. If a standard font is used on a double-density screen, it is pixel-doubled so that it is the same size as when shown on the single-density display.

PalmSource Confidential

The standard font resource type is `'afnt'` on ARM.

To create an ARM font resource, use the XML-based tools provided in the PDK. (You can find more information on the XML-based tools in the book *Building PRC Interfaces*.)

Listing 6.3 shows an XML definition of part of a symbol font resource using the old-style font format, which only supports single density. (Normally, font resources contain many more glyphs than are shown here.) Table 6.5 describes its elements.

Listing 6.3 Standard font resource

```
<FONT_RESOURCE RESOURCE_ID="10000">
  <FONT_TYPE> 0x9000 </FONT_TYPE>
  <FONT_ASCENT> 9 </FONT_ASCENT>
  <FONT_DESCENT> 1 </FONT_DESCENT>
  <FONT_GLYPHS>
    <FONT_GLYPH>
      <FONT_GLYPH_CODE> 0x0003 </FONT_GLYPH_CODE>
      <FONT_GLYPH_IMAGE>
        "....##....."
        "...###....."
        "..######.#."
        ".#######.#."
        "########.#."
        "########.#."
        ".#######.#."
        "..######.#."
        "...###....."
        "....##....."
      </FONT_GLYPH_IMAGE>
    </FONT_GLYPH>
    <FONT_GLYPH>
      <FONT_GLYPH_CODE> 0x0004 </FONT_GLYPH_CODE>
      <FONT_GLYPH_IMAGE>
        "....##...."
        "....###..."
        "#.######.."
        "#.#######."
        "#.########"
        "#.########"
        "#.#######."
        "#.######.."
        "....###..."
        "....##...."
      </FONT_GLYPH_IMAGE>
```

```
    </FONT_GLYPH>
  </FONT_GLYPHS>
  <FONT_MISSING_GLYPH>
    <FONT_GLYPH_IMAGE>
      "....."
      "#####"
      "#...#"
      "#...#"
      "#...#"
      "#...#"
      "#...#"
      "#...#"
      "#####"
      "....."
    </FONT_GLYPH_IMAGE>
  </FONT_MISSING_GLYPH>
</FONT_RESOURCE>
```

Table 6.5 Standard font elements

Element	Description
FONT_TYPE	Indicates the font resource format and usage; it should be set to 0x9000 for Palm Latin fonts. For subfonts, the value should be 0x9400.
FONT_ASCENT	The distance in pixels from the top of the font rectangle to its baseline. For subfonts, use the same value as you used in the font map.
FONT_DESCENT	The distance in pixels from the baseline to the bottom of the font rectangle. For subfonts, use a value greater than or equal to the value provided in the font map.

Table 6.5 Standard font elements *(continued)*

Element	Description
FONT_GLYPHS	A list of one or more FONT_GLYPH elements, one for each glyph in the font. There is a maximum of 256 glyphs in a font. A font does not have to define all 256 glyphs.
FONT_GLYPH	Maps a character code (or part of a character code) with its glyph. Each FONT_GLYPH element contains a FONT_GLYPH_CODE element followed by a FONT_GLYPH_IMAGE element.
FONT_GLYPH_CODE	The character code for the glyph. For double-byte subfonts, this is the low byte of the double-byte character. FONT_GLYPH elements must be specified in increasing order of FONT_GLYPH_CODE.
FONT_GLYPH_IMAGE	The bitmap image for the glyph. Each row of the bitmap is specified as a quoted string. The number of rows defined for each glyph must be equal to the font height (the ascent plus the descent). In each row of the bitmap, a character in the quoted string represents one pixel. The “off” pixels are represented by '.' characters; the “on” pixels are represented by '#' characters. The width of the glyph is defined by the number of characters in the row. Each row in the glyph must have the same width.

PalmSource Confidential

Table 6.5 Standard font elements *(continued)*

Element	Description
`FONT_MISSING_GLYPH`	Defines the glyph bitmap image that is used for any undefined characters in the font. The `FONT_MISSING_GLYPH` has one element, which is a `FONT_GLYPH_IMAGE` element, with the same format as in `FONT_GLYPH` elements described above. Note that each subfont within a hierarchical font still ends with a missing glyph image.

Extended Font Resources

Palm OS 5 supports extended font resources (`'afnx'` on ARM) for each of the default system fonts. An extended font resource defines at most 256 characters, but it contains a separate set of glyphs for each possible screen density. Currently, the only supported densities are single (160x160), 1.5X or "QVGA" (240x320), and double (320x320). When Palm OS draws text to the screen, it first determines the screen density and then uses the glyphs for that density.

To create an ARM extended font resource, use the XML-based tools provided with the PDK. (You can find more information on the XML-based tools in the book *Building PRC Interfaces*.)

Listing 6.4 on page 64 shows the XML definition of a small extended font resource. (Normally, extended font resources contain many more glyphs than are shown here.) This resource is similar to the standard font resource, but has these notable differences:

- The `FONT_TYPE` element should be 0x9200 for a Palm Latin font and 0x9600 for a Palm Shift JIS subfont.
- The ascent and descent are single-density metrics.
- There is a separate set of glyphs (`FONT_GLYPHS` elements) for each supported screen density. The `FONT_EXTENDED_ITEM` defines the glyphs for one density.

This element consists of a FONT_DENSITY element followed by a FONT_GLYPHS element. The FONT_DENSITY may be 72 (single density), 108 (1.5X or QVGA density), or 144 (double density). These numbers roughly correspond to the density's dots-per-inch.

The font extended item elements must be defined in order of increasing density. You ***must*** include the single density glyphs, and then optionally either the 1.5X or the 2X.

The set of glyphs must be identical in all densities of the font. For example, if a glyph for the letter 'A' is defined in the normal density font, it must also be defined in the double-density font, and vice versa.

The normalized dimensions of each glyph bitmaps must be identical in all densities of the font. For example, if the glyph for the letter 'A' has width 10 in the normal density font, it must have the width 20 in the double-density font.

Note that because the double density font has metrics that are exactly double those of the single-density font, the double-density font always has even values for its font metrics. If you design the double-density glyphs before the single-density glyphs, be sure to use even values for the width and height.

Listing 6.4 Extended font resource

```
<FONT_EXTENDED_RESOURCE RESOURCE_ID="10003">
  <FONT_TYPE> 0x9200 </FONT_TYPE>
  <FONT_ASCENT> 8 </FONT_ASCENT>
  <FONT_DESCENT> 0 </FONT_DESCENT>
  <FONT_EXTENDED_ITEMS>
    <FONT_EXTENDED_ITEM>
      <FONT_DENSITY> 72 </FONT_DENSITY>
      <FONT_GLYPHS>
        <FONT_GLYPH>
          <FONT_GLYPH_CODE> 0x0001 </FONT_GLYPH_CODE>
          <FONT_GLYPH_IMAGE>
            "..........."
            ".....#....."
            "....###...."
            "...#####..."
            "..#######.."
            ".#########."
```

```
            "###########"
            "..........."
         </FONT_GLYPH_IMAGE>
      </FONT_GLYPH>
      <FONT_GLYPH>
         <FONT_GLYPH_CODE> 0x0002 </FONT_GLYPH_CODE>
         <FONT_GLYPH_IMAGE>
            "..........."
            "###########"
            ".#########."
            "..#######.."
            "...#####..."
            "....###...."
            ".....#....."
            "..........."
         </FONT_GLYPH_IMAGE>
      </FONT_GLYPH>
   </FONT_GLYPHS>
   <FONT_MISSING_GLYPH>
      <FONT_GLYPH_IMAGE>
         "....."
         "#####"
         "#...#"
         "#...#"
         "#...#"
         "#...#"
         "#####"
         "....."
      </FONT_GLYPH_IMAGE>
   </FONT_MISSING_GLYPH>
</FONT_EXTENDED_ITEM>
<FONT_EXTENDED_ITEM>
   <FONT_DENSITY> 144 </FONT_DENSITY>
   <FONT_GLYPHS>
      <FONT_GLYPH>
         <FONT_GLYPH_CODE> 0x0001 </FONT_GLYPH_CODE>
         <FONT_GLYPH_IMAGE>
            "......................"
            "......................"
            "......................"
            "..........#..........."
            ".........###.........."
            "........#####........."
            ".......#######........"
            "......#########......."
            ".....###########......"
            "....#############....."
            "...###############...."
```

```
            "..#################..."
            ".###################.."
            "#####################."
            "......................"
            "......................"
        </FONT_GLYPH_IMAGE>
    </FONT_GLYPH>
    <FONT_GLYPH>
        <FONT_GLYPH_CODE> 0x0002 </FONT_GLYPH_CODE>
        <FONT_GLYPH_IMAGE>
            "......................"
            "......................"
            "#####################."
            ".###################.."
            "..#################..."
            "...###############...."
            "....#############....."
            ".....###########......"
            "......#########......."
            ".......#######........"
            "........#####........."
            ".........###.........."
            "..........#..........."
            "......................"
            "......................"
            "......................"
        </FONT_GLYPH_IMAGE>
    </FONT_GLYPH>
</FONT_GLYPHS>
<FONT_MISSING_GLYPH>
    <FONT_GLYPH_IMAGE>
        ".........."
        ".........."
        "##########"
        "##########"
        "##......##"
        "##......##"
        "##......##"
        "##......##"
        "##......##"
        "##......##"
        "##......##"
        "##......##"
        "##########"
        "##########"
        ".........."
        ".........."
    </FONT_GLYPH_IMAGE>
```

```
        </FONT_MISSING_GLYPH>
      </FONT_EXTENDED_ITEM>
    </FONT_EXTENDED_ITEMS>
  </FONT_EXTENDED_RESOURCE>
```


PalmSource Confidential

Index

B

boldFont 45, 47

D

DAL
    BigDAL 5
    Globals 19
    Small DAL 3
    SmallDAL 4
Dynamic Heap 9, ??–17, 17–??, 24, 29
dynamic input area 35

G

GenerateXRD 48
Globals 19
    Blitter 32
    DAL 13
    HAL 20
    Identifying Hardware Features 30
    Low Memory Globals 13, 16
    Screen Manager 33
    Setting from Card Header 28

H

HAL 3
    Globals 20

I

input area 35

K

kernel 19

L

largeBoldFont 46, 47
largeFont 45, 47
ledFont 46, 47

M

Memory 12
    Cache 11
    Configuring 14, 15
    Direct Mapped I/O Configuration 14
    HwrPreRAMInit() 15
    Initial Memory Map 12
    New Page Tables 16
    RAM 12
    Refined Memory Map 13
    Regions 9
    Scaling Heap Space 11
    Virtual Memory Map 11
MMU 11, 12, 13, 16

P

Page Tables 16
    First-level 16
    Second-level 16
PalmRC 48

R

ROM
    Building 6
    Installing into RAM 7
    SmallROM 4
ROM Builder 6, 25

S

Scatterload files 3
stdFont 45, 47
Storage Heap 9, 12–17, 24, 29
symbol11Font 45, 47
symbol7Font 46, 47
symbolFont 45, 47

W

widebin 6