



Palm OS 5 ARM Programming

Palm OS[®] 5 SDK (68K) R3

CONTRIBUTORS

Written by Brian Maas

Engineering contributions by Owen Emry, Bob Ebert, David Berbessou, David Fedor, and Greg Wilson

Copyright © 1996 - 2003, PalmSource, Inc. and its affiliates. All rights reserved. This documentation may be printed and copied solely for use in developing products for Palm OS® software. In addition, two (2) copies of this documentation may be made for archival and backup purposes. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form or by any means or used to make any derivative work (such as translation, transformation or adaptation) without express written consent from PalmSource, Inc.

PalmSource, Inc. reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of PalmSource, Inc. to provide notification of such revision or changes.

PALMSOURCE, INC. AND ITS SUPPLIERS MAKE NO REPRESENTATIONS OR WARRANTIES THAT THE DOCUMENTATION IS FREE OF ERRORS OR THAT THE DOCUMENTATION IS SUITABLE FOR YOUR USE. THE DOCUMENTATION IS PROVIDED ON AN "AS IS" BASIS. PALMSOURCE, INC. AND ITS SUPPLIERS MAKE NO WARRANTIES, TERMS OR CONDITIONS, EXPRESS OR IMPLIED, EITHER IN FACT OR BY OPERATION OF LAW, STATUTORY OR OTHERWISE, INCLUDING WARRANTIES, TERMS, OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND SATISFACTORY QUALITY. TO THE FULL EXTENT ALLOWED BY 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, OR PUNITIVE DAMAGES OF ANY KIND, OR FOR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THIS DOCUMENTATION, EVEN IF PALMSOURCE, INC. OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

PalmSource, the PalmSource logo, AnyDay, EventClub, Graffiti, HandFAX, HandMAIL, 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, PalmPak, PalmPix, PalmPoint, PalmPower, PalmPrint, Palm.Net, Simply Palm, ThinAir, and WeSync 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 DOCUMENTATION IS PROVIDED ON A COMPACT DISC, THE OTHER SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT ACCOMPANYING THE COMPACT DISC.

Palm OS 5 ARM Programming
Document Number 9001-003
July 30, 2003
For the latest version of this document, visit
<http://www.palmos.com/dev/support/docs/>.

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

Table of Contents

Palm OS 5 ARM Programming	1
Understanding Palm OS 5 and ARM	1
Palm Application Compatibility Environment	2
Using ARM Subroutines	3
Calling ARM Subroutines	4
Writing ARM Subroutines	4
Isolate the Performance-Critical Area in Your 68K Application .	5
Convert the ARM Subroutine to Take One Argument	6
Handle 68K and ARM Technical Differences	6
Test the ARM Subroutine	8
Build the ARM Subroutine	9
Overview of Included Files	10
ARM Subroutine Sample Files	10
Windows DLL Sample Files	11
CodeWarrior Project Sample Files	11
Index	13

Palm OS 5 ARM Programming

This document is not intended for all Palm OS® application developers.

Most Palm OS 5 applications do not need native ARM code and will not benefit from using native ARM code. Palm OS 5 itself runs as ARM code, so all API functions run at the full speed of the ARM processor. If you have an application that performs adequately on Palm OS 5, then you do not need to write ARM native code.

However, some application algorithms may benefit from being rewritten in native ARM instructions. This document is intended for developers who have applications that require a performance improvement in order to perform adequately on Palm OS 5.

Understanding Palm OS 5 and ARM

Palm OS 5 is a complete port of the Palm operating system from a 68K processor to an ARM processor.

68K:

The term **68K processor** refers to the family of Motorola 68000 processors.

ARM:

The term **ARM processor** refers to the family of Advanced RISC Machine processors. An ARM processor is a type of 4-byte RISC processor, and is available from many sources.

Starting in Palm OS 5, the entire operating system runs natively on the ARM processor. When an application calls a Palm OS API function, the API function runs at the full speed of the ARM processor.

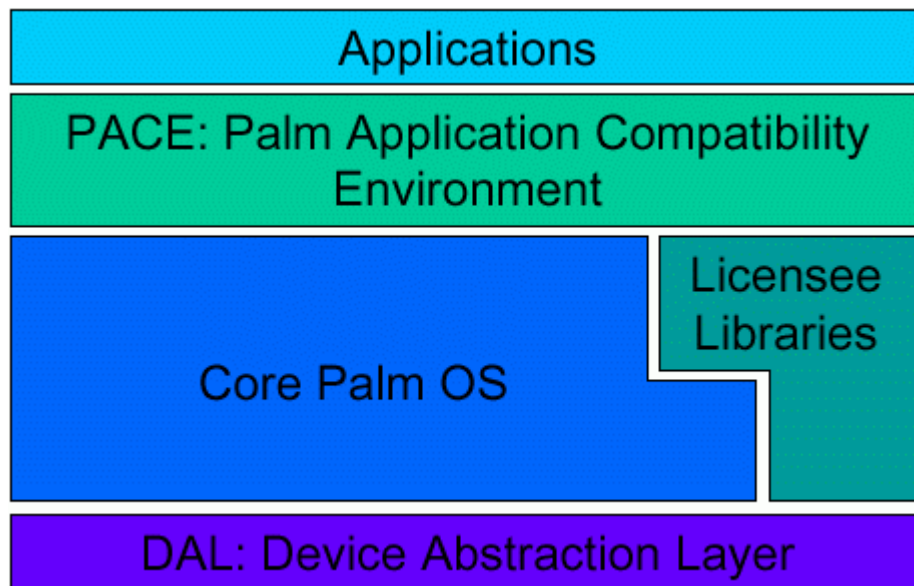
Palm Application Compatibility Environment

Palm OS 5 also provides the Palm Application Compatibility Environment (**PACE**) on ARM. PACE allows existing 68K applications to run on the ARM processor in an emulation mode.

Because Palm OS functions are native and not emulated, PACE provides excellent performance for most 68K applications. As a result, most 68K applications will not benefit significantly from being rewritten for ARM.

[Figure 1](#) shows how PACE provides a compatibility layer between 68K applications and Palm OS 5 running natively on ARM.

Figure 1 Palm Application Compatibility Environment



Because an application's 68K code is emulated in PACE, certain algorithms—such as those performing data encryption or compression—may benefit from being rewritten in native ARM instructions.

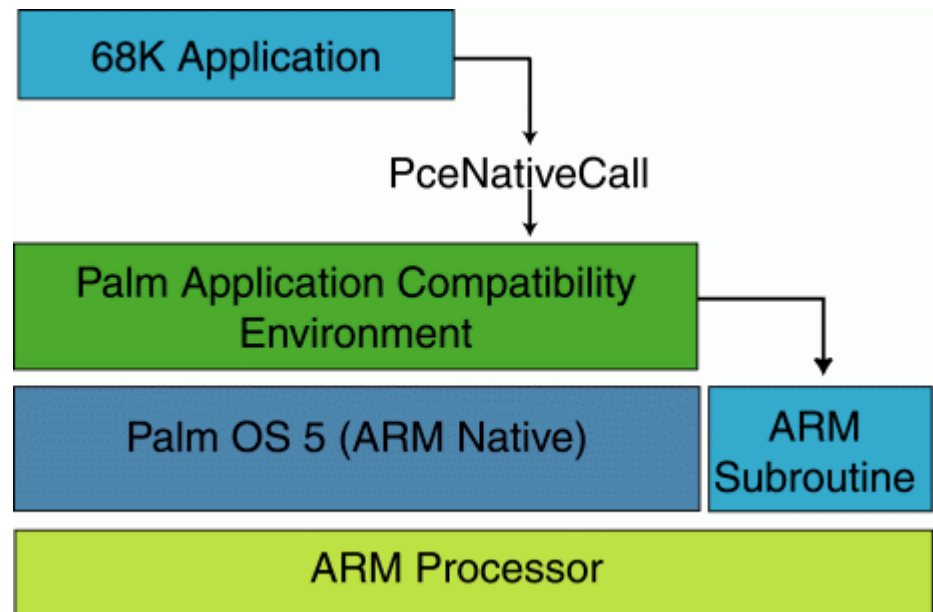
Using ARM Subroutines

If you have a processor-intensive 68K algorithm, writing an ARM subroutine may improve the performance of your 68K application on Palm OS 5.

An **ARM subroutine** is not a self-contained application; it is a native ARM function that the 68K application can call as a subroutine. The ARM subroutine allows the application to use the full processing power of the ARM hardware.

[Figure 2](#) shows how your 68K application calls your ARM subroutine.

Figure 2 Using PceNativeCall to Call an ARM Subroutine



Calling ARM Subroutines

To call an ARM subroutine from your 68K application, you use the new function `PceNativeCall`. `PceNativeCall` is fully documented in *Palm OS Programmer's API Reference*.

The `PceNativeCall` function takes two arguments:

1. A pointer to the ARM subroutine, generally but not necessarily stored in a code resource.

If the ARM subroutine is stored in a resource, the 68K application can simply lock the resource with the appropriate type and ID to get a pointer to the ARM subroutine.

2. A pointer to a data block, allowing the 68K application to exchange data with the ARM subroutine.

Before calling the ARM subroutine, the 68K application must check that it is running on PACE and must also check the processor type.

In general, a 68K application should provide a 68K subroutine implementing code equivalent to the ARM subroutine in case the application is running on non-ARM hardware (that is, on a handheld running an earlier version of Palm OS). By including both a 68K version and an ARM version of the subroutine, you continue to support the existing Palm Powered™ handhelds as well as Palm OS 5 handhelds. For more information about application compatibility, see *Palm OS Programmer's Companion*.

Writing ARM Subroutines

The ARM subroutine needs to include the proper prototype and the ARM code you want it to contain. A sample ARM subroutine, called `armlet-simple.c`, is included in the Palm OS 5 SDK to show the minimum amount of code required.

The ARM subroutine can also call Palm OS API functions, and can call back into 68K code. The file `armlet-oscalle.c`, also included in the Palm OS 5 SDK, provides an example of calling a Palm OS API function.

The following sections explain the steps for writing an ARM subroutine:

1. “[Isolate the Performance-Critical Area in Your 68K Application](#)” on page 5
2. “[Convert the ARM Subroutine to Take One Argument](#)” on page 6
3. “[Handle 68K and ARM Technical Differences](#)” on page 6
4. “[Test the ARM Subroutine](#)” on page 8
5. “[Build the ARM Subroutine](#)” on page 9

Isolate the Performance-Critical Area in Your 68K Application

To decide which algorithms will benefit from being written as an ARM subroutine, you should start by doing a performance analysis of your 68K application. If your 68K application runs “fast enough” when you do your performance testing, then there is no reason to write an ARM subroutine.

- Test your 68K application using Palm OS Simulator. Palm OS Simulator is the easiest and best way to test your application for Palm OS 5 compatibility. Running your application on Palm OS Simulator will show you whether any algorithms behave differently on Palm OS 5.

Any algorithms that do extensive calculations, such as data encryption or compression, may run slower on Palm OS Simulator. If you notice a performance difference, then you have found a candidate algorithm that might benefit from being rewritten as an ARM subroutine.

- Test your 68K application using the profiling version of Palm OS Emulator. The profiling version of Palm OS Emulator monitors your application’s execution, generating statistics that show which algorithms take the most time.

Emulator can help you pinpoint slow algorithms, but performance on Emulator will not indicate performance on Palm OS 5. Emulator does not include the Palm OS 5 PACE component, but Simulator does.

Convert the ARM Subroutine to Take One Argument

The function `PceNativeCall`, which you use to call an ARM subroutine from your 68K application, takes only two arguments: a pointer to the ARM subroutine and a pointer to a data block. As a result, it will be easier to write your subroutine if it takes a single input argument. For more information about using the `PceNativeCall` function, see *Palm OS Programmer's Companion*.

Handle 68K and ARM Technical Differences

When implementing the ARM subroutine, you should be aware of how the 68K processor and the ARM processor are different. The following sections describe some technical considerations that you need to handle in your ARM subroutine:

- [“Big Endian and Little Endian”](#)
- [“Integer Alignment”](#)
- [“Structure Packing”](#)

Big Endian and Little Endian

The 68K processor uses big-endian integers; the ARM processor uses little endian. **Big** and **little** refer to the order in which the bytes are stored in a multi-byte integer. In big-endian integers, the most significant byte is the first; in little-endian integers, the most significant byte is the last byte.

This means 2- and 4-byte integers are stored in reverse byte order, and thus must be byte-swapped when exchanged between the ARM and 68K processors. Endianness is only relevant in the context of 2- and 4-byte integers (including pointers). Other types of data, such as strings, don't need to be byte-swapped.

PACE automatically byte-swaps the `PceNativeCall` function's `userData68KP` argument, so it can be de-referenced immediately from with the ARM function with no modification. PACE also automatically byte-swaps the 4-byte return value that is passed back to the calling function.

PACE doesn't byte-swap any of the data pointed to by the `userData68KP` argument because PACE doesn't know anything

about what kind of data is being passed. (Remember, only 2- and 4-byte integers need to be byte-swapped, and the `userData68KP` argument is simply a pointer to arbitrary data.)

Byte-Swapping Macros for Use in ARM Subroutines

`Endianutils.h` contains convenience macros to byte-swap 2- and 4-byte integers in your ARM subroutine:

`ByteSwap16(integer)`

Byte-swaps a 2-byte (16-bit) integer value.

`ByteSwap32(integer)`

Byte-swaps a 4-byte (32-bit) integer value.

ARM subroutines are responsible for byte-swapping integers in the data block as necessary.

Integer Alignment

ARM processors require that 4-byte integers be aligned on a 4-byte boundary. 68K processors require only even address (2-byte) alignment.

To handle integer alignment differences, you have the two following options:

1. Allocate data using `MemPtrNew`, carefully declaring data structures with appropriate integer alignment.

`MemPtrNew` always returns a 4-byte aligned address, so you can be sure that the data starts on a 4-byte boundary.

However, you must also be careful that the data itself is properly aligned. When aligning data objects, recognize that 68K and ARM processors align 4-byte objects differently, as shown in [Table 1](#).

Table 1 Default Data Object Alignment

Data Object Size	68K Processor Alignment	ARM Processor Alignment
1 byte	Any address	Any address
2 bytes	2-byte alignment (even address)	2-byte alignment (even address)
4 bytes	2-byte alignment (even address)	4-byte alignment (address is a multiple of 4)

If a 4-byte data object is not properly aligned, the ARM processor may attempt to access the object using an address that is a multiple of 4, resulting in a loss of data.

2. Copy 4-byte integers into local variables before using them.

`Endianutils.h` contains convenience macros that you can use to read and write 4-byte values to and from local variables while simultaneously byte-swapping them:

- `Read68KUnaligned32(address)`

Reads a value from a specified address.

- `Write68KUnaligned32(address, value)`

Writes a specified value to a specified address.

Structure Packing

Different compilers handle the automatic padding of structures differently. Some compilers automatically add padding bytes to align structures on a given byte boundary depending on the compiler options specified. Use care when declaring structures, or make a local copy of any structure that you use.

Test the ARM Subroutine

The ARM subroutine will run on Palm OS 5 on ARM hardware. However, Palm OS Simulator does not run ARM code. Instead, Simulator provides an implementation of Palm OS 5 running on Microsoft Windows. As a result, to test your ARM subroutine on

Simulator, you need to build the subroutine as a Windows DLL. Simulator's implementation of PACE is built to recognize a subroutine call as a call into a DLL.

The Palm OS 5 SDK includes a sample Microsoft Visual C++ project that builds a DLL with one entry point which has the same function as the sample ARM subroutine also included in the Palm OS 5 SDK.

When calling a DLL, the first argument passed to `PceNativeCall` is a pointer to the name of a DLL and the name of the entry point within that DLL that is to be executed, separated by a null character and terminated with a null character (for example, a pointer to the character string `"test.dll\0EntryPoint"`).

By default, Simulator will look for the DLL in the directory where `PalmSim.exe` is running. If you want to place the DLL in a different location, you should specify the full path of the ARM subroutine DLL name (for example, `"c:\\projects\\armletdll\\test.dll\0EntryPoint"`).

Your 68K application should check the processor type:

- If the processor is ARM, the 68K application should call the ARM subroutine.
- If the processor is Windows, the 68K application should call the Windows DLL.

Otherwise, the 68K application should call the 68K version of the subroutine, which assumes the application is running on an earlier version of Palm OS.

Build the ARM Subroutine

You will need to use an ARM compiler to build the ARM subroutine. Palm, Inc. does not provide or support an ARM compiler or development environment, but several are available, such as ARM Developer Suite (ADS) and gcc.

The compiled object file for the ARM subroutine must be linked with the 68K application as a raw binary file. For calculating address offsets, it is generally easiest to put the entry point first in the raw binary file.

Adding ARM Subroutines to a Metrowerks Project

For developers who are accustomed to using Metrowerks, an easy way to include an ARM subroutine is to load the binary ARM instructions into a resource. You can do this with CodeWarrior by using a resource file (a file with the file type `.r`), or by using PilRC. The Palm OS 5 SDK includes a sample project for a sample `ARMCode.r` file.

Overview of Included Files

The following ARM programming samples are included as part of the Palm OS 5 SDK.

ARM Subroutine Sample Files

[Table 2](#) shows the sample files that call ARM code from a 68K application.

Table 2 Calling ARM from 68K Sample Files

Filename	Purpose
armlet-simple.c armlet-simple.bin	A trivial ARM subroutine showing how to pass a pointer from a 68K application.
armlet-oscall.c armlet-oscall.bin	An ARM subroutine showing you how to call a Palm OS API function, using <code>MemPtrNew</code> as an example.
armlet- endianness_and_alignment.c armlet- endianness_and_alignment.bin	An ARM subroutine showing you how to make sure your data is correctly 4-byte aligned.
endianutils.h	Macros for doing endian byte-swapping and 4-byte alignment correction. Used by the <code>armlet-oscall.c</code> and <code>armlet-endianness_and_alignment.c</code> files.
example_data_type.h	Example showing a user-defined structure. Used by the <code>armlet-endianness_and_alignment.c</code> file.

Windows DLL Sample Files

[Table 3](#) on page 11 table shows the sample files that you can use to build a DLL for testing an ARM subroutine with Palm OS Simulator. For background information, see “[Test the ARM Subroutine](#)” on page 8.

Table 3 Windows DLL Sample Files - For Testing with Palm OS Simulator

Filename	Purpose
Simple.dsp	Visual Studio project file for building a DLL file.
SimNative.cpp	The main DLL source file.
SimNative.h	Header file which defines the exports from the DLL file.
StdAfx.cpp	C++ source file used to build a precompiler header file and precompiled types file.
StdAfx.h	Header file used by StdAfx.cpp.

CodeWarrior Project Sample Files

[Table 4](#) table shows the sample files that you can use with Metrowerks CodeWarrior. For background information, see “[Adding ARM Subroutines to a Metrowerks Project](#)” on page 10.

Table 4 Metrowerks CodeWarrior Project Sample Files

Filename	Purpose
<code>sample_project.mcp</code>	CodeWarrior project file.
<code>Starter.c</code>	Source file for the 68K application that calls the ARM subroutine.
<code>Starter.rsrc</code>	Resource file, used for including the ARM subroutine as a code resource.
<code>StarterRsc.h</code>	Header file used by the resource file.
<code>ARMCode.r</code>	Resource file containing the ARM subroutine as a code resource.

Index

Symbols

.r file 10

Numerics

68000 processor 1

68K processor, definition 1

A

ADS 9

Advanced RISC Machine 1

API function PceNativeCall 4

ARM compilers 9

ARM Developer Suite 9

ARM gcc compiler 9

ARM processor, definition 1

ARM programming samples 10

ARM subroutine, definition 3

ARMCode.r 10, 12

armlet-endianess_and_alignment.c 10

armlet-oscill.c 4, 10

armlet-simple.c 4, 10

automatic padding 8

B

big endian 6

building ARM subroutines 9

building Windows DLLs 9

byte boundary alignment 7

byte swapping 6

ByteSwap16 macro 7

ByteSwap32 macro 7

C

calling ARM subroutines 4

calling Palm OS API functions 4

concepts 1

E

Emulator

 see Palm OS Emulator 5

endianness 6

endianutils.h 7, 10

example_data_type.h 10

exchanging data 4

G

gcc compiler 9

I

included files 10

integer alignment 7

L

little endian 6

local variables 8

locking resource 4

M

Microsoft Visual C++ 9

Motorola processor 1

P

PACE, definition 2

Palm Application Compatibility Environment 2

Palm OS 5 1

Palm OS Emulator 5

Palm OS Simulator 5, 8

PceNativeCall 4

performance considerations 2, 5

PilRC 10

R

R file 10

Read68KUnaligned32 macro 8

resource file 10

S

sample ARM subroutine 4

sample_project.mcp 12

simnative.cpp 11

simnative.h 11

simple.dsp 11

Simulator

 see Palm OS Simulator 5, 8

Starter.c 12
Starter.rsrc 12
StarterRsc.h 12
stdafx.cpp 11
stdafx.h 11
structure packing 8

T

testing ARM subroutines 8

U

userData68KP 6
using an ARM code resource file 10
using ARM subroutines 3

V

Visual C++ 9

W

Write68KUnaligned32 macro 8
writing ARM subroutines 4