RAGHUNATH GANNAMARAJU
Palmist: A Tool to Log Palm System Activity
(Under the direction of SURENDAR CHANDRA)

With the proliferation in the usage scenarios of mobile handheld devices, understanding typical client usage patterns is fundamental to developing new policies to improve the usability of these devices. In this thesis, we describe a Palm system call logging tool called Palmist. Palmist allows the practitioner to selectively collect statistics such as the system call invoked, application that invoked the system call, the time of the call and the system call arguments. Our logging mechanism consumes about 20 bytes of memory on the PDA to store the log record. The logging mechanism adds a latency of about 10 msec to collect the log. The mechanism has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 88% (735 of 834 relevant system calls) of the Palm OS 3.5 system calls. Our system can be utilized by system developers to customize their application behavior to optimize system parameters such as energy consumption and ease of use.

INDEX WORDS: Palm OS, Mobile User Access Pattern, Thesis (academic)

PALMIST: A TOOL TO LOG PALM SYSTEM ACTIVITY

by

RAGHUNATH GANNAMARAJU

B.E., Birla Institute of Technology, India, 1997

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2001

© 2001

Raghunath Gannamaraju

All Rights Reserved

PALMIST: A TOOL TO LOG PALM SYSTEM ACTIVITY

by

RAGHUNATH GANNAMARAJU

Approved:

Major Professor: Surendar Chandra

Committee: David Lowenthal
Eileen T. Kraemer

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
December 2001

ACKNOWLEDGMENTS

I would like to thank Dr. Surendar Chandra for all the help and guidance he has provided during the course of this thesis. I am grateful for his help during some of the debugging sessions. I would also like to thank Dr. David Lowenthal and Dr. Eileen Kraemer for consenting to serve on my committee.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . iv

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

CHAPTER

1 INTRODUCTION . . . 1

2 BACKGROUND . . . 4

2.1 TRAP HANDLING . . . 4

2.2 FEATURE MANAGER . . . 5

2.3 PALM DATABASES . . . 6

2.4 HOTSYNC . . . 6

3 SYSTEM ARCHITECTURE . . . 7

3.1 OBJECTIVES . . . 7

3.2 PALMIST TRAP HANDLER INSTALLATION MECHANISM . . . 7

3.3 PALMIST TRAP HANDLERS . . . 9

3.4 DOWNLOADING PALM'S ACTIVITY LOG ONTO THE DESKTOP . . . 14

3.5 EXPERIMENTAL SETUP . . . 14

4 PERFORMANCE RESULTS . . . 15

4.1 LOG COLLECTION OVERHEAD . . . 15

4.2 PALMIST USAGE EXAMPLES . . . 20

5 RELATED WORK 27

6 CONCLUSIONS AND FUTURE DIRECTIONS 29

BIBLIOGRAPHY 31

APPENDIX

A PSEUDO CODE FOR INSTALLING PALMIST TRAP FUNCTION (SYSTRAP 0XA00E) 33

B PSEUDO CODE FOR THE LIBRARY INITIALIZATION FUNCTION 34

C PSEUDO CODE FOR A TYPICAL SYSTEM CALL TRAP HANDLER 35

D PSUEDO CODE FOR COMMON LIBRARY 37

E PSEUDO CODE FOR THE CONDUIT TO DOWNLOAD LOGS TO THE DESKTOP 40

List of Tables

3.1 Distribution of the number of parameters to Palm OS 3.5 system calls logged with Palmist 10
3.2 Palm event latency (rounded down) 12
4.1 Palmist overhead per system call 17
4.2 Relative overhead for Palmist internal system calls 17
4.3 Time taken for typical operation using Address Book application 18
4.4 Time taken for typical operation using Calculator application 19
4.5 Time taken for typical operation using Date Book application 19
4.6 Pinemark Benchmarks 23

LIST OF FIGURES

2.1 Palmist patch to system traps 5

3.1 Palmist mechanism to trap system calls 8

4.1 “Representative” session 22

4.2 Text Pinemark benchmark (Note: Calls marked with a * were also popular in the “representative” session) 25

4.3 System calls invoked with time (Text Pinemark benchmark) 26

CHAPTER 1

INTRODUCTION

There is an explosion in the number and variety of handheld devices. Dataquest estimates that 3.55 million handheld devices were delivered worldwide during the first quarter of 2001. Palm OS based [8] PDAs account for 87% of the handheld devices sold [13]. Falling prices and the increasing availability of newer applications is driving the popularity of hand held devices.

Hand held devices are constrained in their usability by the available network, battery capacity and display characteristics. The typical client usage pattern has a profound impact on the usability and the efficacy of system management policies. For example, the access pattern of the user significantly affects the battery energy consumption. Depending on the idle duration, the Palm device consumes more energy to shut down and power up than to remain idle [4]. A policy that conserves energy by shutting down the unit after a fixed idle period would not be effective if the user resumes work immediately, forcing a powerup. Also, it has been shown [3] that the batteries perform better under pulsed discharge, rather than a steady discharge. Policies that can change the computation to leverage these device characteristics can be expected to prolong battery life. Understanding the actual user behavior is fundamental to designing these newer policies.

Representative client usage logs are invaluable in answering realistic questions about the system behavior of PDAs. However, little is known of the actual usage behavior of hand held devices. Earlier work [5, 4] used oscilloscopes to capture system activity. Such probes restrict the ability to collect mobile user activity. Also, publicly available Palm

OS Emulator (POSE) [6] is widely used for debugging and profiling Palm applications. These emulation programs typically run on desktop computers. Hence, their applicability to profile and capture mobile users' behavior is limited. There is a need to instrument the actual devices to capture the mobile user behavior.

The primary goal of this work is to develop tools to capture the system behavior of PDA devices. The logs collected can be used by practitioners to evaluate the performance of their system design. For example, operating system developers for the PDA platform can analyze the efficacy of their energy saving features for a typical user. Application designers can also improve the general usability by improving the accessibility of popular features.

In this work, we focus our attention on Palm devices. We describe Palmist, a tool to log system calls for the Palm device. One of the design goals in developing Palmist was to capture enough system activity details to replay them in log play back tools. The Palms were instrumented using the *hack* [9] mechanism to collect statistics such as the system call invoked, application that invoked the system call, the time of the call and the system call arguments. The system allows the practitioner to selectively log a subset of interesting system calls. The collected information can be periodically *sync*ed to a desktop computer for further processing. These instrumented Palms can be distributed to *typical target users* and their usage patterns collected over a period of time. The trends in the users' access patterns can be analyzed.

One of the challenges in designing Palmist was to constrain the overhead of the additional instrumentation. The overhead introduced by Palmist depends on the amount of system call activity logged. We show that our logging mechanism adds an acceptable amount of overhead in collecting the system call events. On the average, the log records consume about 20 bytes of storage on the Palm device. The Palmist trap handlers add an additional latency of about 10 msec per system call logged. Depending on the application scenario, we show that in the worst case, a fully instrumented Palm can run up to two

orders of magnitude slower. However, disabling a few popular calls and using a faster PDA for interactive tasks adds an extra 50% overhead only. The mechanism also has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 88% (735 of 834 of relevant system calls) of the Palm OS system calls.

The remainder of this report is organized as follows: in Chapter 2 we begin by setting the background for our work by briefly describing the Palm OS features used in developing our system. We describe the Palmist tool in detail in Chapter 3. Experimental setup to measure the logging overhead and the evaluation methodologies used in our study are described in Chapter 4. Chapter 5 places our work in the context of other related work in the field. We present our conclusions and outline future research directions in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we briefly describe several features of Palm OS that were exploited in developing Palmist. In particular, we describe how the Palm OS system handlers can be replaced with our own trap handlers, how the *feature manager* can be utilized to provide global memory regions, how Palm stores permanent data in databases, as well as the HotSync operation to upload data from the Palm to a desktop computer.

2.1 TRAP HANDLING

The Palm OS supports a trap mechanism to service system events (such as "user entered a Graffiti character", "system timer ticked", "power-off", etc.). The system events are serviced through a system-wide trap handler vector (illustrated in Figure 2.1). Each system call has a 16-bit Trap ID value associated with it. By default, when an event occurs or a system call is executed, a lookup is done in the trap handler vector and the appropriate Palm OS system call trap handler is executed. Based on their functionality, the system calls can be broadly categorized into various APIs. Some of the API categories include Form API, Database API, Resource Manager API etc. Palm OS 3.5 supports a total of 881 system calls.

Palm OS allows users to modify the default system behavior by installing their own trap handlers. These user defined trap handlers are generally known as *hacks*. These *hacks* are in the form of Palm resource files (.prc) of type *HACK* and contain the user-defined handler code (as a code resource) along with the trap ID of the system call being handled.

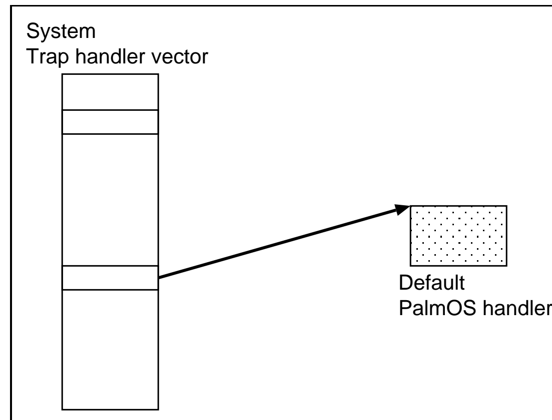


Figure 2.1: Palmist patch to system traps

Together, the file type and the trap ID are used to get the location of the user-defined handler. If a hack is activated, the user-defined code is executed in lieu of the original Palm OS provided trap handler.

2.2 Feature Manager

Palm OS application's global data is not generally available outside the application. *Feature* memory provides access to data that persists between invocations of an application. *Feature* memory is allocated from the storage heap. The values stored in *feature* memory persist until the device is reset or until the memory is explicitly freed. Applications can share data by publishing a *feature* that contains a pointer to the shared data.

2.3 Palm Databases

Palm databases function similarly to a file storage abstraction. All Palm OS data, including RAM and ROM applications, user data, and application preferences are stored in databases. The Palm database is composed of unstructured records. There is no inherent limit on database size (as long as it fits into memory). However, records within a database are restricted to about 64 KB. Record databases are a particular advantage during HotSync operations: records can be marked as new, unchanged, or changed. Unchanged records can be omitted from the synchronization with status flags.

2.4 HotSync

HotSync [7] is the process of synchronizing the data on the Palm and desktop computer. During the HotSync process, data can be backed up to the desktop from Palm and data from the desktop can be uploaded into the Palm device. Palm allows per application customization of the synchronization operation using *conduits*. Each application on the Palm has a conduit associated with it. New conduits are installed by specifying the conduit DLL, along with the associated Palm database. These conduits run on the desktop computer, synchronizing the application's database on the Palm with data on the desktop.

Chapter 3

System Architecture

3.1 Objectives

The primary goal in designing the Palmist system was to develop the functionality required to capture information about system events. We wanted to capture enough information so that play-back tools can be developed to recreate a particular user interaction. Palmist instruments the system calls to collect the following per-call information:

- system call invoked,
- application that invoked the system call,
- the start time of the call,measured in elapsed ticks since last reset (1 tick = 10 msec)
- the system call arguments. Pointers are dereferenced up to one level of indirection.

3.2 Palmist trap handler installation mechanism

We utilized the *hack* mechanism to develop our logging tool. We developed a unique system call specific *hack* to collect these logs. Tools such as *hackmaster* [9] can manage the activation and deactivation of the *hack* extensions to the Palm OS system software. However, there are limitations on the number of *hacks* handled by hackmaster. Hackmaster application can manage a maximum of 50 system calls only. This can be increased by modifying the Hackmaster application. But, hackmaster application's source code is not available. Since Palm OS 3.5 provides 881 system calls, we developed our own

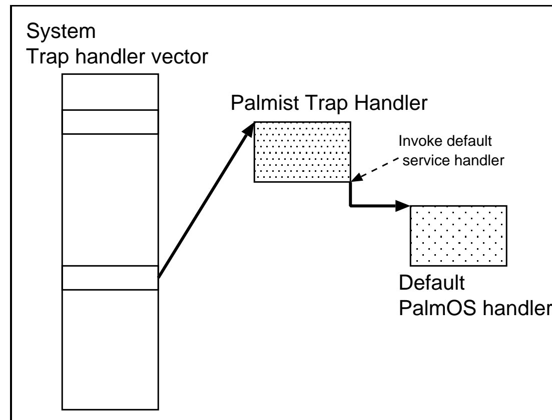


Figure 3.1: Palmist mechanism to trap system calls

strategy to install these user defined trap handlers. Throughout the rest of this report, we refer to these *hacks* as Palmist trap handlers.

A pseudo code fragment, which illustrates our approach to installing Palmist trap handlers, is shown in Appendix A. We utilized the Palm OS system calls *SysGetTrapAddress* and *SysSetTrapAddress* to develop our system. For a given trap ID, *SysGetTrapAddress* returns the address of the original system trap handler, while *SysSetTrapAddress* can be used to change the system trap handler's address. Palmist trap handlers maintain the original functionality of the system by storing the original trap handler (returned from *SysGetTrapAddress*) before updating the system trap vector using *SysSetTrapAddress*. We utilize the *feature manager* facility *FtrSet* to store the original system trap handler address. After the log collection, the original system call is invoked from within our logging function (Figure 3.1).

3.3 Palmist trap handlers

A unique Palmist trap handler is installed for each system call that needs to be logged. These Palmist trap handlers have the same function prototype as that of the original system call. The function parameters to the original system call are passed on to the Palmist trap handler.

In order to reduce the overall code size for all the unique Palmist trap handlers, we utilize a common logging library function from these trap handlers. In the Palm OS, libraries need to be loaded before they can be invoked. In order to reduce the logging overhead of loading and opening the library for every system call (quantitatively measured in Section 3.3.1), we initially loaded and opened the library once while installing the Palmist trap handlers (Appendix B). Subsequent invocations directly call this (already loaded and opened) library function. Throughout the data logging, this library is never closed. As a result of this, one less library can be opened than the number of libraries that could be opened without Palmist.

A pseudocode segment illustrating a per system call Palmist handler is illustrated in Appendix C. Each trap handler passes the system call's trap ID as well as a packed argument buffer to the common logging library function. While packing the arguments, pointers are dereferenced up to one level for *string* and *struct* data structures. The distribution of the number of parameters to system calls are tabulated in Table 3.1. Of the 735 system calls that can be instrumented with Palmist, 276 system calls do not return a value.

A pseudocode detailing our common logging library function is illustrated in Appendix D. The common logging library function logs the system call parameters along with the current system call start time (in 10 msec ticks since last reset) and the application ID of the active application. Application ID is the database ID of the current application's resource database. Application ID is unique between Palm resets. Each application has a set of resources associated with it with each resource having a name.

No.of parameters	No.of system calls
0	89
1	211
2	183
3	110
4	57
5	38
6	9
7	6
8	1
9	4
10	0
11	2
12	0
13	1

Table 3.1: Distribution of the number of parameters to Palm OS 3.5 system calls logged with Palmist

The application names associated with each application ID can be accessed directly from the 'tAIN' resource during Palmist initialization. We use a unique feature ID as a global flag to prevent stack overflow while logging those system calls which are used in the logging process. We set this flag when we enter logging function and reset it while exiting. Logging function is executed only when flag is reset.

The common logging library function stores the system call log information in a Palm database. This storage log database is initially created during the installation process (Appendix B). In order to reduce the latency in opening this log database, we open the database once during this initialization and store the database ID of this database as a *feature*. Subsequently, Palmist reuses this database ID to open the database.

The common logging library function appends new log information into this storage database. Each Palm record has an associated meta-data of 8 bytes. In order to minimize

the storage requirements of this per-record meta-data, we pack the logs (by appending to the last record) into records that can be up to 64 KB (Palm OS limitation). New log data are always appended to the last record; the position at which the next data item is to be written is stored in the first two bytes of the last record in the database. When the last record fills up, a new record is created. On average, at the end of log collection, 32KB is wasted in the last record.

Such logging mechanisms introduce additional data collection overhead. As we will note later in Section 3.3.2, system restrictions allow us to only capture about 88% (735 of 834 relevant system calls) of the system calls using Palmist trap handlers.

3.3.1 Palmist trap handler optimizations

System call logging introduces additional overhead to the normal operation of the Palm device. In order to identify the expensive operations in our log collection strategy, we profiled the entire Palmist code. We measured the difference in system time before and after a particular operation. On the Palm device, the system clock can be measured with a granularity of 10 msec ticks.

Some of the interesting system calls utilized by the Palmist trap handler mechanism along with their execution times (measured in 10 msec ticks) are tabulated in Table 3.2. It should be noted that the actual Palm system calls consume between 0 and 1 clock ticks. From Table 3.2, we note that the library load and open calls are expensive; taking between 2 and 3 ticks.

The library load and open calls were optimized by initially preloading the library during the Palmist installation. Palmist Trap Handlers reuse this preloaded library to reduce the logging overhead.

Also, with *DmOpenDatabaseByTypeCreator*, the system searches for the particular database and hence is less efficient. We replaced the *DmOpenDatabaseByTypeCreator* with *DmOpenDatabase*. *DmOpenDatabase* needs the database ID instead of the database

Event	**Latency (ticks) (tick=10 msec)**
libload	2.85
libopen	1.12
MemAlloc	0.208
Pack 1 parameter	0.001
Pack 2 parameters	0.002
GetApplicationID	0.015
SysGetTicks	0.003
DmOpenDatabaseByTypeCreator	0.62
DmOpenDatabase	0.33
FtrGet	0.04

Table 3.2: Palm event latency (rounded down)

name to open the particular Palm database. This database ID is unique between system resets. We locate this ID during the Palmist trap handler installation and store the value as a *feature*. This database ID is queried from the *feature manager* before opening the database using *DmOpenDatabase*. We profiled the time taken by *DmOpenDatabaseByTypeCreator*, *DmOpenDatabase* and *FtrGet* From Table 3.2 we note that this database optimization reduces the function overhead from 6.2 msec to 3.7 msec.

3.3.2 Log collection limitations

The Palmist system call collection mechanism has inherent limitations in collecting the activity of certain system calls. Several system calls (low level memory functions) are also indirectly referenced by other system calls used by Palmist. For example, we utilize the system call *DmCloseDatabase* within the common logging library (Appendix D). Instrumenting *DmCloseDatabase* causes stack overflow; resulting in a system crash.

Several system calls are called exclusively during the HotSync process. Since we disable logging during the HotSync operation (as described in Section 3.4), these calls cannot be logged.

Also, Palm applications have access to limited stack space. Instrumenting system calls necessitates additional stack space. Application developers can increase this default stack space. Since we do not modify the actual applications directly, some system calls which call other system calls repeatedly cause a system stack overflow. Hence, logging was turned off for such system calls.

A breakdown of the system calls that are not being logged into categories depending on why they are not being logged is given below.

- Stack Overflow : 4
- HotSync : 30
- Low Memory Access : 21
- Chunk Underlocked : 2
- Chunk Overlocked : 2
- Palm Crashes : 62
- Obsolete Calls : 17
- Feature Manager : 8

Some of the system calls that are not logged because of these limitations include *MemHandleNew*, *MemHandleLock*, *MemHandleUnlock*, *MemHandleFree*, *MemCardInfo*, *TimGetTicks*, *SysCurAppDatabase*, *DmOpenDatabase*, *DmNumRecords*, *DmNewRecord*, *DmWrite*, *DmReleaseRecord*, *DmGetRecord* and *DmCloseDatabase*. We are currently exploring ways to log some of these system calls using alternative techniques.

3.4 Downloading Palm's activity log onto the desktop

We utilized the HotSync process to download the logs from the Palm to the desktop [7]. We developed a conduit to download the system call log data collected by Palmist. We utilized the *SyncManager* API to develop this conduit. The pseudo code describing our conduit is illustrated in Figure E. During the HotSync process, the log storage database is opened to upload data to the desktop. The conduit downloads the data record by record from the Palm storage database. Since the database is already opened by the conduit process, it is not possible to collect system logs while HotSyncing the logs. Palmist traps check for this condition and disable logging as long as the system is performing a HotSync of the log database. The collected statistics are written into a text log file on the host desktop. The records are then deleted from the Palm database to free up storage resources.

3.5 Experimental setup

Palmist was developed to collect system call information for Palm OS 3.5 [1]. Palm OS 3.5 supports a total of 881 system calls. As we will describe in Section 3.3.2, Palmist can only collect access logs for 735 system calls. In general, Palmist can be extended to collect the logs for newer calls introduced in Palm OS 4.0. We developed the Palm conduits using Conduit SDK 4.0.2. We used Code warrior R6 on a Windows NT desktop to develop Palmist. We've tested Palmist on IIIxc, IIIxe and m505 Palm computing devices.

Chapter 4

Performance Results

In the last chapter, we described the Palmist tools and how it collects log information about system call activity. In this chapter, we present our results and experiences in using the tool as well as the overhead in collecting such usage logs. We describe the memory and log collection latency overheads in using Palmist. We show the measured overhead for targeted micro-benchmarks. We also outline a usage scenario to highlight how Palmist can be utilized to identify performance bottle-necks in Palm applications. We present the system calls invoked for a typical usage session of Palm as well as a session involving Pinemark benchmark [14] Text test.

4.1 Log collection overhead

First, we describe the storage overhead in collecting logs. Then we describe the HotSync overhead in downloading the logs. Next we utilize our own microbenchmarks to experimentally measure the per system call latency in collecting the system activity.

4.1.1 Log collection storage overhead

Palmist's installation on Palm entails memory overhead, both for the code of Palmist trap handlers as well as the logs that are collected using Palmist system. Each Palmist trap handler code occupies about 400 to 700 bytes of memory (depending on the number of arguments to the particular system call) along with another 2.53 KB for the common logging library function. The Palmist system code consumed about 700 KB for all the

735 Palmist trap handlers. It should be noted that a typical Palm device offers about 7.7 MB of free storage (before installing additional software).

The system call information itself is logged in a log database on the Palm device. Such logging consumes the limited amount of memory available in a Palm device. Each log record consumes at least 12 bytes of storage space to log the Trap ID, system time (in 10 msec ticks since the system reset) and the application ID. The system call parameters consume additional storage space. The size of the entire log is limited by the amount of memory available in the system. Each record in the Palm database has an additional overhead of 8 bytes to store meta-data. Hence, we pack multiple logs into a single Palm record to amortize this storage cost.

When all the available memory is consumed by the logging database, the system crashes for lack of memory for essential system operations. Hence, Palmist disables logging when the available memory is lower than a configurable threshold. For our experiments, we chose a threshold of 512 KB. Palmist checks to see if the available memory is above this threshold before storing any logs. Once the logs fillup the available memory, the user must HotSync the logs to the desktop before further logging is possible.

4.1.2 HOTSYNC LATENCY

Uploading the logs from the Palm to the desktop adds additional latency during the HotSync process. For example, HotSyncing a 1064 KB log from a Palm IIIxe to a desktop with dual 933MHz Pentium III and 512 MB memory using a serial cable connected at 115 kbps adds about 2 minutes to the HotSync process.

System call	Latency w/o Palmist (in 10 msec ticks)	Latency w/ Palmist (in 10 msec ticks)
WinClipRectangle	7	921
WinSetDrawWindow	11	1169
RctCopyRectangle	3	921
WinGetDrawWindow	3	915
FntSetFont	6	1171

Table 4.1: Palmist overhead per system call

Operation	System call	Time (in msec)
	OriginalCall	0.47
Log record	TimGetTicks	0.03
	CurrAppDatabase	0.15
Memory functions	MemHandleNew	0.66
	MemHandleFree	0.57
	MemHandleLock/ MemHandleUnlock	0.85
	MemCardInfo	1.05
	Total Time for Memory Operations	3.13
Storage functions	FtrGet	0.39
	DmOpenDatabase	3.30
	DmWrite	2.82
	DmGetRecord	0.47
	DmReleaseRecord	0.41
	Total Time for Storage Operations	7.39

Table 4.2: Relative overhead for Palmist internal system calls

Logging of System calls disabled	Time on Palm IIIxe (in seconds)	Time on Palm m505 (in seconds)
None disabled	240	110
Top 3 disabled	200	85
Top 7 disabled	180	80
Top 10 disabled	130	67
Without Palmist	30	30

Table 4.3: Time taken for typical operation using Address Book application

4.1.3 LOG COLLECTION LATENCY

Palmist trap handlers add additional logging overhead to Palm system calls. In this section, we measure the per system call overhead as well as the typical slowdown that Palmist causes on the Palm devices instrumented with it.

First, the latency introduced by Palmist for each system call was measured using five popular system calls that were invoked during the usage sessions described in Section 4.2. These system calls were micro-benchmarked by invoking each system call with and without the Palmist trap handlers a 1000 times. The times taken are tabulated in Table 4.1. From Table 4.1, we note that even with the optimizations outlined in Section 3.3.1, the log overhead is quite high. The relative overhead due to Palmist depends on the time taken by the original system call to execute. A simple system call which executes in less time has higher relative overhead due to Palmist and vice versa. The overhead added for each of the system calls used by Palmist while instrumenting a system call *DmWrite* is shown in Table 4.2. From Table 4.2, we note that the log storage functions and memory functions are expensive, consuming as much as 1500% and 665% of the original system call, respectively.

Logging of System calls disabled	Time on Palm IIIxe (in seconds)	Time on Palm m505 (in seconds)
None disabled	65	28
Top 4 disabled	45	21
Top 5 disabled	45	20
Top 10 disabled	32	18
Without Palmist	10	10

Table 4.4: Time taken for typical operation using Calculator application

Logging of System calls disabled	Time on Palm IIIxe (in seconds)	Time on Palm m505 (in seconds)
None disabled	118	45
Top 5 disabled	75	36
Top 10 disabled	60	30
Without Palmist	20	20

Table 4.5: Time taken for typical operation using Date Book application

However, the actual slowdown experienced by a Palm user depends on the applications' system call vs. application code mix and also on the interactiveness of the applications.Typical Palm applications are highly interactive with a low ratio of system call time to application time. In order to measure the actual impact of Palmist on Palm usage with regard to time, we measured the time taken to perform typical operations; add a new contact using the *Address Book* application, perform a simple calculation using the *Calculator* application and add a new appointment into the *Date Book*. The user interaction time depends on the familiarity of the user with the Palm device. A user who is less familiar with the graffiti input mechanism is expected to take more time to input data. These experiments were repeated by instrumenting a newer Palm m505 (which uses a 33 MHz processor as compared to a 20 MHz processor used in the Palm IIIxe). It should

be noted that m505's run Palm OS 4.0. However, Palm OS 4.0 is backward compatible and can still run Palm OS 3.5 based Palmist. Our goal was to investigate the effectiveness of a faster processor in reducing the logging overhead. For these interactive tasks, the faster processor's speed is unnoticeable when the Palm is not instrumented. The times for various interactive operations are tabulated in Tables 4.3, 4.4, 4.5. From Table 4.3, we observe a slowdown of upto 8 times for *Address Book* operation while using Palm IIIxe. A faster m505 reduces the Palmist trap handler overhead for the same operation to about 3.5 times. Disabling the Palmist trap handlers for popular system calls shows improvement; disabling the ten most popular system calls along with utilizing a faster PDA reduces the overhead to about 150% of the original operation for *Date Book* operation. Disabling of system call logging should be done selectively, depending on the user data that needs to be collected. Even with system call logging disabled for a few of the system calls the user actions can be interpreted from the system calls logged.

These results show that a combination of a faster PDA and selective disabling of logging of popular system calls can aid in limiting the logging latency to acceptable levels.

4.2 Palmist Usage Examples

In the previous section, the overhead due to Palmist was analyzed. In this section, two usage scenarios that highlight the applicability of Palmist in collecting Palm system activity are explored. Depending on developers' needs, Palmist can be tailored to collect the statistics for a few interesting system calls (rather than all calls possible with Palmist). Palmist was used to analyze a representative user interaction session with a Palm device. Palmist was used to analyze the Pinemark benchmark program also.

The representative user session was comprised of the following actions on an instrumented Palm.

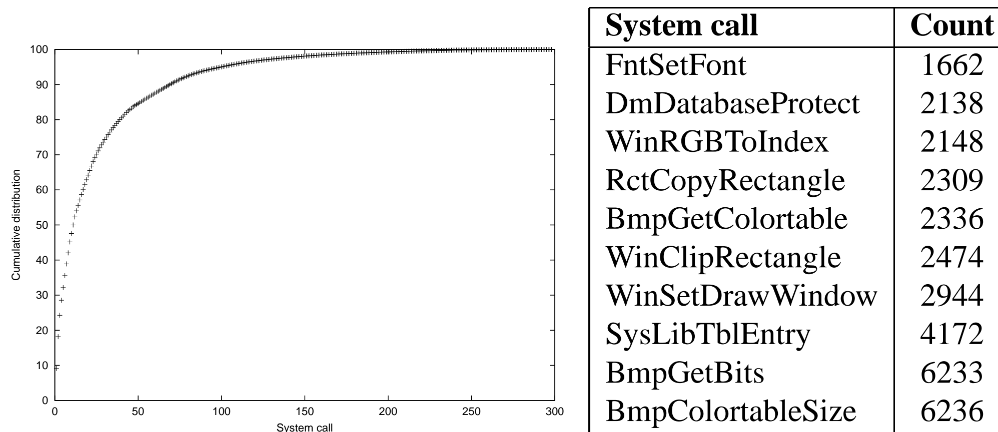
1. Launch Address Book application

- Add a contact (Last name: Foo, First name: Bar, Company: FooBar, Work: 1234567) to the ADDRESS BOOK
- Remove a contact address (Accessories, Palm Computing contact) from the ADDRESS BOOK

2. Launch Calculator
   - Make a calculation ($500000 \div 12$)
3. Launch Datebook
   - Switch to next day's schedule
   - Make an appointment ("Meeting" at 10:00)
4. Hotsync to download data to desktop

The collected logs were then uploaded to the desktop and analyzed. During our "typical" user session, a total of 68608 system calls were invoked. The total amount of data collected from this typical user session was 1.34 MB. We plot the invocation frequency of the system calls as a cumulative distribution in Figure 4.1(a). From Figure 4.1(a) we note that not all system calls were invoked during this user session; only 296 system calls (of 735 instrumented calls) were invoked. We also note that a few system calls were quite popular; 10 calls were invoked 50% of the time. We tabulate these ten most popular system calls in Figure 4.1(b). We also plot the frequency of argument sizes to system calls in Figure 4.1(c). Most of the system calls have no arguments. On average, the argument buffer consumes 8 bytes of storage.

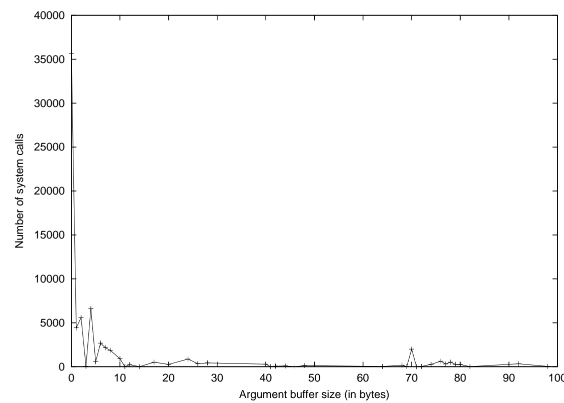
With an average per system call storage requirements of about 20 bytes, logging the popular system calls can easily fill up the available storage space on the Palm device. For more popular system calls, statistical sampling may be utilized to reduce the amount of



System call	Count
FntSetFont	1662
DmDatabaseProtect	2138
WinRGBToIndex	2148
RctCopyRectangle	2309
BmpGetColortable	2336
WinClipRectangle	2474
WinSetDrawWindow	2944
SysLibTblEntry	4172
BmpGetBits	6233
BmpColortableSize	6236

(a) Cumulative distribution of the number of times a system call was invoked

(b) Occurrence frequency of ten most popular system calls



(c) Distribution of system call argument size (in bytes)

Figure 4.1: “Representative” session

Type of Test	Benchmark w/o Palmist (in Pinemarks)	Benchmark w/ Palmist (in Pinemarks)
Text test	831.5	88634.6
Database test	151	2468.1
Math test	58.5	429

Table 4.6: Pinemark Benchmarks

log data. Depending on developers' needs, Palmist can be tailored to collect the statistics for a few interesting system calls (rather than all calls possible with Palmist).

Next we utilized Pinemark [14], a publicly available Palm benchmark to measure the slowdown in instrumenting as many system calls as possible. Pinemark measures the performance in pinemark units which roughly indicates the amount of time taken to perform certain categories of operations. Pinemark supports three benchmark categories:

- **Text:** The text benchmark tests text display functionality,
- **Database:** The database benchmark tests the performance of database operations such as open, read, write and close.
- **Math:** The math benchmark analyzes the arithmetic performance.

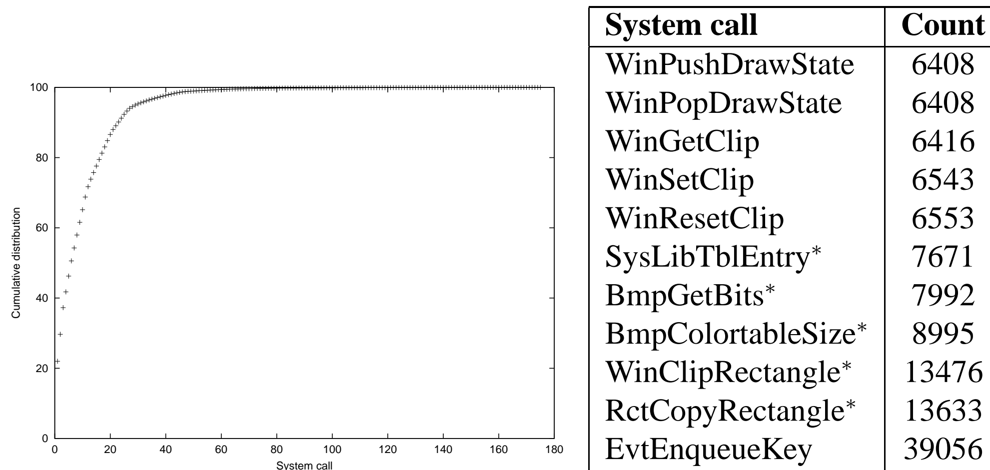
Pinemark was used first without and then with Palmist installed on Palm. The results (averaged over 3 trials) are tabulated in Table 4.6. From Table 4.6, we note that the Text benchmarks are significantly affected by the Palmist trap handlers. Math and Database benchmarks display a moderate slowdown. Most of the system calls exercised by the Text system call were instrumented with Palmist trap handlers. Since the Text benchmark repeatedly exercises these instrumented system calls, the slow down is pronounced. On

the other hand, most of the math related operations do not use systems calls. Hence, the Math test shows minimal slowdown.

During the Text benchmark, the system collected 3.05 MB of trace logs. The Palmist system traced 185,361 systems calls, 177,589 of which were directly invoked by the benchmark. From Figure 4.2(a), we note that the ten most popular system calls account for over 65% of the system calls. The ten most popular system calls for the Text benchmark are listed in Figure 4.2(b). We observe that *SysLibTblEntry*, *BmpGetBits*, *BmpColortable-Size*, *WinClipRectangle* and *RctCopyRectangle* are among most frequently called system calls both in our typical usage session as well as the Text test of Pinemark benchmarking program. The argument buffer size (shown in Figure 4.2(c)) is similar to our “representative” session.

Next we utilize Palmist to show how a developer might use Palmist to understand the performance bottlenecks within the Text benchmark. We plot the system calls invoked (trap ID) with respect to the time of invocation in Figure 4.3. The developer will notice the frequent breaks in system activity (for example, around 1050 secs). They can further explore this particular time interval in greater detail.

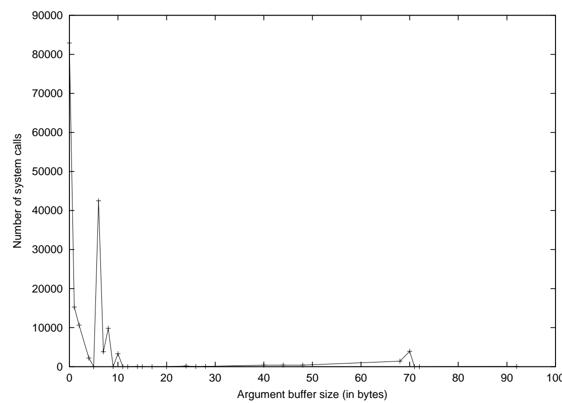
In this section, we described some of the usage scenarios for the Palmist tool. We used the tool to identify popular system calls and the system calling sequence for a mobile user. These calling patterns could help the Palm system developers to selectively optimize these “typical” and popular calling sequences. Techniques such as [11] can be employed to optimize the Palm code. The application developers can also utilize this information to better customize their applications. They can tune these optimizations based on actual mobile user behavior, rather than using a simulated session.



System call	Count
WinPushDrawState	6408
WinPopDrawState	6408
WinGetClip	6416
WinSetClip	6543
WinResetClip	6553
SysLibTblEntry*	7671
BmpGetBits*	7992
BmpColortableSize*	8995
WinClipRectangle*	13476
RctCopyRectangle*	13633
EvtEnqueueKey	39056

(a) Cumulative distribution of the number of times a system call was invoked

(b) Occurrence frequency of eleven most popular system calls



(c) Distribution of system call argument size (in bytes)

Figure 4.2: Text Pinemark benchmark (Note: Calls marked with a * were also popular in the "representative" session)

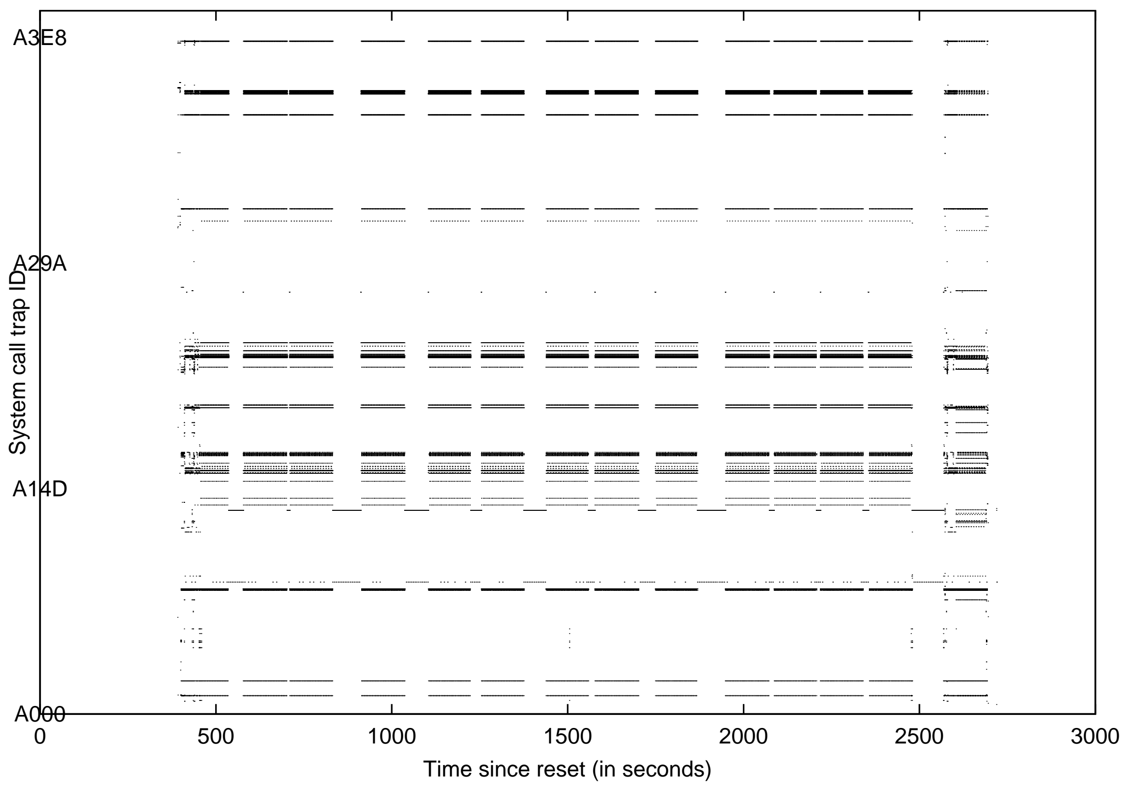


Figure 4.3: System calls invoked with time (Text Pinemark benchmark)

CHAPTER 5

RELATED WORK

A number of earlier efforts have explored various techniques to capture system behavior. Tools such as SimpleScalar [2] and SimplePower [15] are cycle accurate execution driven simulators. In the case of Simplepower, a framework that includes a cycle-accurate energy model for memory and bus subsystems was described. In this paper, code from video and signal processing domains was analyzed for the purposes of energy estimation. Simplepower can be used to identify energy utilization hotspots and enable developers and compiler designers to focus their optimization efforts towards those areas.

Tools such as POSE [6] emulate the handheld systems and can be utilized to profile a typical user session. Because this work uses Palm OS based PDAs, Palm OS Emulator was of particular interest. Palm OS Emulator, POSE for short, is software that emulates the hardware of various Palm OS based PDAs. This software needs ROM image, which contains the Palm OS operating system code, in order to emulate a particular PDA. This emulation program runs on Windows, Mac OS and Unix platforms. POSE is very useful during the development phase for applictions on the Palm OS platform. POSE can be used to debug and profile the applications on Palm devices. It was extensively used for debugging purposes in development of the Palmist.

Other research efforts such as [5, 4, 12] have used hardware probes to capture system usage information. However, such techniques have limited applicability in a mobile scenario. Mobile users can be expected to use their devices differently while on the road compared to a controlled setting near tethered and bulky simulators and capture devices.

Newman, et al. [12] outline an approach very similar to our Palmist approach. However, they abandoned this approach (because of time constraints) for a microbenchmark based technique to measure the power consumption characteristics of the Palm device.

The VTrace logging tool [10] for windows clients is close in spirit to Palmist. VTrace was developed to obtain time-stamped traces of certain activities in Windows NT and Windows 2000 to study energy management techniques for laptop computers. In order to study the effects of varying the CPU voltage and clock speed and of powering down various system components, it is necessary to know when power-consuming components (such as the CPU, the disk, and the network interface card) were active and what they were doing at each instant. In general, the system development and profiling tool support for mobile PDAs lag their desktop computer counterparts. Palmist system logging has higher data collection overhead as compared to VTrace due to limited processing power and storage of the mobile device.

CHAPTER 6

CONCLUSIONS AND FUTURE DIRECTIONS

We have described a Palm system activity logging tool called Palmist. Palmist allows the practitioner to selectively collect statistics such as the system call invoked, application that invoked the system call, the time of the call and the system call arguments. We show that the overhead introduced by Palmist depends on the amount of system call activity logged. We show that our logging mechanism adds an acceptable amount of overhead in collecting the system call events. On average, the log records consume about 20 bytes of storage on the Palm device. The Palmist trap handlers add an additional latency of about 10 msec per system call logged. We show that a fully instrumented Palm can run up to two orders of magnitude slower. However, disabling a few popular calls and using a faster PDA adds only 50% extra overhead for interactive tasks. The mechanism also has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 88% (735 of 834 relevant system calls) of Palm OS system calls. Palmist can be utilized by system practitioners to study the effects of their policy decisions on mobile users.

Several extensions to the Palmist tool can be implemented. Alternate strategies to collect information about system calls that cannot be directly logged by Palmist can be explored.

Ways to incorporate a mechanism to play back system behavior logs through the publicly available Palm emulator can be implemented. This would allow the developers the ability to repeat actual user behavior. The emulator can also be augmented with per system

call energy consumption values. Such an augmented emulator can be utilized to evaluate energy-aware policies that exploit the application-system interaction. For example, for a Palm user, the system might choose different power saving policies depending on the usage pattern. Power management policies that can exploit these access behaviors can be developed and verified.

BIBLIOGRAPHY

[1] PalmOS SDK Version 3.5. http://www.palmos.com/dev/tech/tools/sdk35.cgi.

[2] Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[3] Carla Fabiana Chiasserini and Ramesh R. Rao. Pulsed battery discharge in communication devices. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking (MOBICOM '99)*, pages 88–95, Seattle, WA, August 1999.

[4] Todd L. Cignetti, Kirill Komarov, and Carla Schlatter Ellis. Energy estimation tools for the palm. In *ACM MSWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[5] Jason Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, February 1999.

[6] Greg Hewgill et al. Palm OS Emulator (POSE). http://www.palmos.com/dev/tech/tools/emulator/.

[7] Palm Inc. Hotsync technology. http://www.palm.com/support/hotsync.html.

[8] Palm Inc. Palm device. http://www.palm.com/.

[9] Edward Keyes. Hackmaster. http://www.daggerware.com/hackmstr.htm.

[10] Jacob Lorch and Alan J. Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000. http://msdn.microsoft.com/msdnmag/issues/1000/VTrace/VTrace.asp.

[11] Robert Muth, Scott Watterson, and Saumya Debray. Code specialization using value profiles. In *Static Analysis Symposium*, pages 340–359, July 2000.

[12] Mark Newman and Jason Hong. A look at power consumption and performance on the 3com palm pilot. http://guir.cs.berkeley.edu/projects/p6/finalpaper.html, May 1998.

[13] NPD INTELECT PDA press update. Top two PDA brand shares. October 2000 - February 2001. http://www.intelectmt.com/corp/intelectmt/press/press-it/press˙010330.htm, March 2001.

[14] Pine tree software. Pinemark benchmark. http://www.pinetreesoftware.net/palmos/pinemark/.

[15] N. Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA*, pages 95–106, 2000.

Appendix A

Pseudo code for installing Palmist trap function (systrap 0xA00E)

```
//Get original trap handler's address
originalFn = SysGetTrapAddress(sysTrapMemHeapCompact);

//Store the address as a feature
FtrSet('A00E', 1000, (UInt32)originalFn);

// Open the database by type ('HACK') and creator ID ('A00E')
DB = DmOpenDatabaseByTypeCreator('HACK','A00E', mode);

//Get the address of Palmist trap handler
memPtr = DmGetResource('code', 1000);
p = MemHandleLock(memPtr);

//Install Palmist trap handler for Trap ID 0xA00E
SysSetTrapAddress(0xA00E, p);

//Free memory resources and close the Palmist trap handler code database
MemHandleUnlock(memPtr);
DmReleaseResource(memPtr);
DmCloseDatabase(DB);
```

Appendix B

Pseudo code for the library initialization function

```
//Load and open the common logging library
e = SysLibLoad('libr', 'TEST', &refNum);
SampleLibOpen(refNum, &clientContext);

//Test for the presence of the database by trying to opening
if (DmOpenDatabaseByTypeCreator('Data', 'Proj', mode) ==
                        DatabaseDoesNotExist)
{
 // The database doesn't exist, create it now.
 DmCreateDatabase(0, "ProjDB", 'Proj', 'Data', false);

 // Open the application's database.
 DB = DmOpenDatabaseByTypeCreator('Data', 'Proj', mode);
 }

// Set the feature with the value of the Database ID
FtrSet('DBID', 1000, (UInt32)dbIDP);
DmCloseDatabase(DB);
```

APPENDIX C

PSEUDO CODE FOR A TYPICAL SYSTEM CALL TRAP HANDLER

```
UInt8 MyTrapHandler (Boolean set, UInt8 value)
{
UInt8 (*oldtrap)(Boolean set, UInt8 value);

//get Original trap Handler Address using Feature Manager
FtrGet('A34B', 1000, \&oldtrap);

//Allocate memory to store the parameters to the system call
memhandle = MemHandleNew(sizeof(set)+sizeof(value));
memptr = MemHandleLock(memhandle);

//Store the parameters passed to the system call in the memory allocated
*((Boolean *) memptr) = set;
*((UInt8 *) ((char *)memptr+sizeof(set))) = value;

//Call the library function which logs the system call and the parameters
SampleLibWriteStatistic(4, 0xA34B, sizeof(set)+sizeof(value), (char *)memptr);

//Free the allocated memory
MemHandleUnlock(memhandle), MemHandleFree(memhandle);
```

```
// Call the original trap handler in order to preserve the
// original system behaviour
return oldtrap(set, value);
}
```

Appendix D

Psuedo code for Common Library

```
// Check if there is enough memory
MemCardInfo (0, 0, 0, 0, 0, 0, 0, &freeBytes);
if(freeBytes <= threshold) return;

// Get the current time in terms of system ticks
ticks = TimGetTicks();

// Get the appID of the current application on whose
//  behalf the trap handler is executing
SysCurAppDatabase (&cardNoP, &appID);

//Open the log database
if (! FtrGet('DBID', 1000, &dbid))
  DB = DmOpenDatabase(0, dbid, mode);

// Get number of records in log database.
// If this is first time this function is being
// called create a record
if ((numrecords = DmNumRecords(DB)) == 0)
    Create a new record
```

```
// Get record pointer of the last record
recHandle = DmGetRecord(DB, numrecords−1);
recText = MemHandleLock(recHandle);

// Get the offset at which data is logged in the record.
// Initially the data is logged at byte offset of 2.
// Subsequent logs are written following the last update.
nRecPos = *(UInt16 *) recText;

// If data cannot fit completely in the current record
// release the current record and create new record
// and start writing from offset 2 in that record
// FIX_LOG_SIZE  = sizeof(trapID)+sizeof(ticks)+
//          sizeof(appID)+argbufsize=12

if ( (nRecPos + FIX_LOG_SIZE + argSize) >= MAX_REC_SIZE)
    Create a new record
// Write the Trap ID whose handler called this function
DmWrite(recText, nRecPos + 0, &trapId, size(trapId));

// Write the system time in ticks
DmWrite(recText, nRecPos + 2, &ticks, sizeof(ticks));

// Write the application ID of the current application
DmWrite(recText, nRecPos + 6, &appID, sizeof(appID));
```

```
// Write the argument size for this call
DmWrite(recText, nRecPos + 10 , &argSize, sizeof(argSize));

// Write the parameters to the system call
if (argSize != 0)
  DmWrite(recText, nRecPos + FIX_LOG_SIZE, argBuf, argSize);

// Update the new write position to
// the byte next to the data just written
nRecPos += FIX_LOG_SIZE + argSize;
DmWrite(recText, 0, &nRecPos, 2);

// Clean up and close the database
MemHandleUnlock(recHandle), DmReleaseRecord(DB, n-1, true);
DmCloseDatabase(DB);
```

Appendix E

Pseudo code for the conduit to download logs to the desktop

```
// Open the database on Palm
SyncOpenDB("ProjDB", 0, handle, eDbRead|eDbWrite);

rInfo.m_pBytes = new BYTE[64000];
rInfo.m_FileHandle = handle;
rInfo.m_TotalBytes = 64000;
rInfo.m_dwReserved = 0;

// Get number of records in the database
SyncGetDBRecordCount(handle, rNumRecs);

// Get each record
for (rInfo.m_RecIndex = 0; rInfo.m_RecIndex < rNumRecs; rInfo.mRec_Index++)
  // Write into log file on Desktop
  SyncReadRecordByIndex(rInfo);

// Purge all the records from the database on Palm
SyncPurgeAllRecs(handle);

// Close the database on Palm
```

```
SyncCloseDB(handle);
```