# Post-mortem Debug and Software Failure Analysis on Symbian OS

Kui Wang

University of Tampere
Department of Computer Sciences
Computer Science
M.Sc. thesis
Supervisor: Jyrki Nummenmaa
January 2007

University of Tampere
Department of Computer Sciences
Computer Science
Kui Wang: Post-mortem Debug and Software Failure Analysis on Symbian OS
M.Sc. thesis, 59 pages, 7 index and appendix pages
January 2007

---

This thesis introduces a debugging utility named Mobile Crash on Symbian OS. Mobile Crash is able to trap Symbian OS program panics and processor exceptions. It further collects program data and processor state that may help to solve the trapped failure, and sends the collected data for analysis. The thesis presents the background knowledge of embedded system development and Symbian OS to help the reader to understand the main topic. The thesis also discusses other ways to do debugging on Symbian OS, which include using emulator and tracing application execution. In addition, the thesis compares different debugging methods and gives general guidelines for applying these methods on Symbian OS development.

Key words and terms: Symbian OS, Embedded System, Debug, Mobile Crash.

# Contents

## Preface

A post-mortem debugging utility that can automate trapping and reporting Symbian OS software failures helps to detect programming errors and provide information to solve those errors. When applying such a utility to software integration and testing phases, statistics can be drawn from the collected error reports to point out the most vulnerable software components.

The study discussed in the thesis and the implementation of the post-mortem debugging utility – Mobile Crash – have been carried out in Nokia Tampere, Symbian Product Platform organization. I would like to thank Erkki Salonen, who is the manager of Low Level software development team, for his support during the time of implementing Mobile Crash and writing the thesis.

I would like to thank the thesis instructor, Professor Jyrki Nummenmaa, for his kind guide to my M.Sc studies at the Department of Computer Sciences, University of Tampere, and for his time of examining the thesis.

My colleagues in the Low Level software development team have cooperated with me to develop Mobile Crash. They are Hannu Pyydysmaki, who has implemented the stack decoding application, Esa Karvanen, who has set up the crash binary file database and Mobile Crash web UI, Oleg Rodionov, who has customized the Symbian crash debugger implementation, Zhigang Yang, who has verified Mobile Crash on various product platforms, and many others, who helped to implement, integrate, and test Mobile Crash. I have enjoyed the team work with them and would like to express my appreciation to their cooperation.

Last but not least, I would like to thank my parents and my wife, Liu Yang, for caring and encouraging me to complete this thesis work.

Kui Wang

10 Dec 2006, Tampere, Finland

# 1. Introduction

This thesis work introduces a debugging utility – Mobile Crash – that can capture, transfer, and analyse Symbian OS software defects on smartphones. The major functions of Mobile Crash are that, first it provides a way to debug Symbian OS software failures during the period when development ends and transmits to field testing, and second, it provides statistic data to evaluate the product maturity. The utility consists of several software components that are installed to smartphones and PCs. The techniques applied to build up these components include Symbian C++ development, Windows C++ development, Database and Web UI development.

In brief Mobile Crash works so when a software failure happens on the smartphone, the debug agent installed to the phone can catch the software failure, collect relevant data of the panicked software component including panicked thread stack, run time loaded binary modules list, and CPU registers. Then Mobile Crash transfers these data either via USB connection to a workstation and further to the server that decodes the data and store to database, or via SMS through the GSM network to a gateway that forwards the data to the same server. Software developers can then browse the decoded data via the web UI, and analyse these data to locate the bug in the panicked software component. The web UI is able to calculate software failure statistics based on the end user's requirement and input. See Figure 1 for an overview of Mobile Crash.

The similar debugging idea has been used, for example, on Window OS as well. Readers may have experienced that when a Windows application crashes, a dialogue is popped up asking whether the end user would like to send a report containing the crash information to Microsoft. The sent reports are then analyzed and statistic is drawn in order to solve the most critical (most often crashed) components to effectively improve the OS stability. [Murphy, 2004]

To better support readers to understand this debugging utility, the basics of embedded system development and Symbian OS are first introduced. The following chapters present two common ways to solve software failures on Symbian OS, which are debugging applications with Symbian emulators, and tracing application execution on target smartphones. After these the thesis continues to present Mobile Crash in different perspectives including design, implementation and testing. It further provides use cases of applying Mobile Crash to debug Symbian OS applications, and examples of analyzing software failure statistics to evaluate the product maturity. While explaining these various ways to debug Symbian OS applications, the thesis also provides a study of the favourable and unfavourable parts of each way and gives the author's own opinion of how and when to apply them on the Symbian software development.

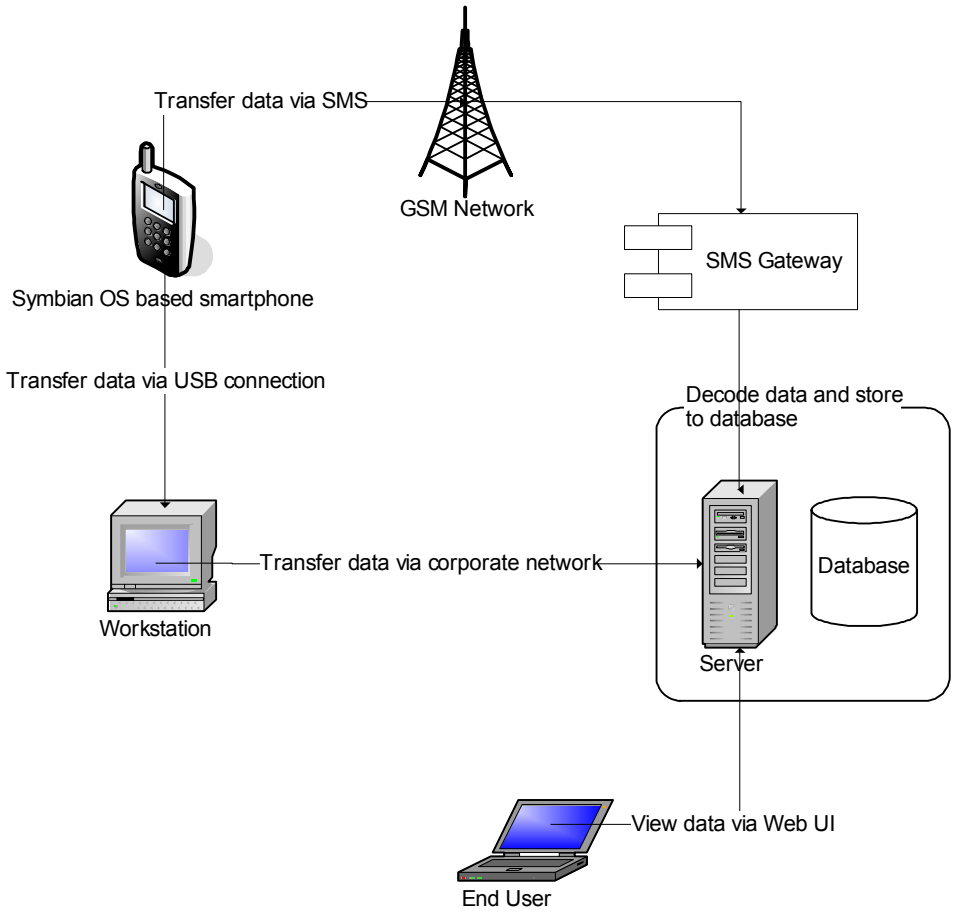


Figure 1 Overview of Mobile Crash

# 2. Basics of Embedded System Development

## 2.1 Characteristics of embedded system

Embedded systems are the kind of devices designed and manufactured to accomplish a few special tasks. They were first developed to facilitate the military and space industry during the 1960's, due to the characteristic of accurate data process by special designed hardware. Embedded systems differentiate with PCs (Personal Computer) in the way that the former are designed for special-purpose tasks and the later are general-purpose computers. In an embedded system the hardware is often just enough to accomplish the defined special tasks, and often no hard disk is integrated. The microcontroller, which consists of a microprocessor and other logic and memory ICs (Integrated Circuit), is the heart of an embedded system. The microcontroller is frequently programmable. [Wikipedia - Embedded System, 2006]

When developing embedded system, cost and performance are usually the most important factors need to be considered. In a volume product development, the cost becomes more important than the performance. The typical consumer electronic embedded systems, such as portable music player and mobile phone, are shipped to market in the volume of millions. Saving a few dollars on each product, and then multiplied by the volume means effectively cutting the production cost. The very-high-volume embedded systems commonly have the configuration of one SoC (System-on-Chip) plus a few other ICs (usually memory chips). The design goal for these embedded systems is to choose a SoC exactly good enough to accomplish the desired functions and use as less memory as possible, to decrease the production cost.

The software developed for embedded systems, especially those not reside in a hard disk but rather a flash memory or being programmed into a programmable IC, are sometimes called firmware, i.e., software embedded in the hardware devices. Embedded systems are expected to keep functioning for continuous months or years, and some of them might be out of the reach of humans, for instance, down to an oil well or out to the space. Therefore the designs of embedded system and firmware need to be reliable and robust, and testing on them need to be carefully planned and executed. Unlike PC software, firmware cannot be patched and any update to the firmware often means reprogram the IC or reload the software to flash memory. The procedure to update firmware is frequently complicated and expensive, and often the customers do not have the expertise or equipments to do the update. Hence, the release of firmware targets on zero error, and once released they commonly do not return to factory for an update. [Wikipedia - Embedded System, 2006]

## 2.2 Embedded system development overview

The embedded system development has been made easier and more cost-effective by the fast-developed SoC technology in recent years. Processor design and architecture have become an intellectual property block, which can be licensed to ASIC (Application-Specific Integrated Circuit) / ASSP (Application-Specific Standard Product) manufactures. The most popular processor design is the ARM architecture by the ARM Ltd (Advanced RISC Machines Ltd). The ARM architecture has been licensed to many well-known silicon chip vendors, such as IBM, Texas Instruments, Philips, Sharp and Samsung. It's estimated that over 75% of all 32-bits embedded processors worldwide are ARM architecture based, at the time when this thesis is written. ARM processors have been widely integrated in all kinds of embedded systems, including Apple iPod products, Nintendo Game Boy Advance consoles and smartphones from Nokia, Siemens and BenQ [Wikipedia - ARM, 2006]. The silicon chip vendors support embedded system development by providing the board support package, which is the platform having processors, memory chips and all kinds of hardware peripherals integrated in a board. The board support package works as a product equivalent system, on which developers can implement and integrate software to hardware, debug and test software. Figure 2 shows the board support package of Texas Instruments OMAP 2420 application processor, which integrates the ARM1136 processor. [OMAP, 2005]

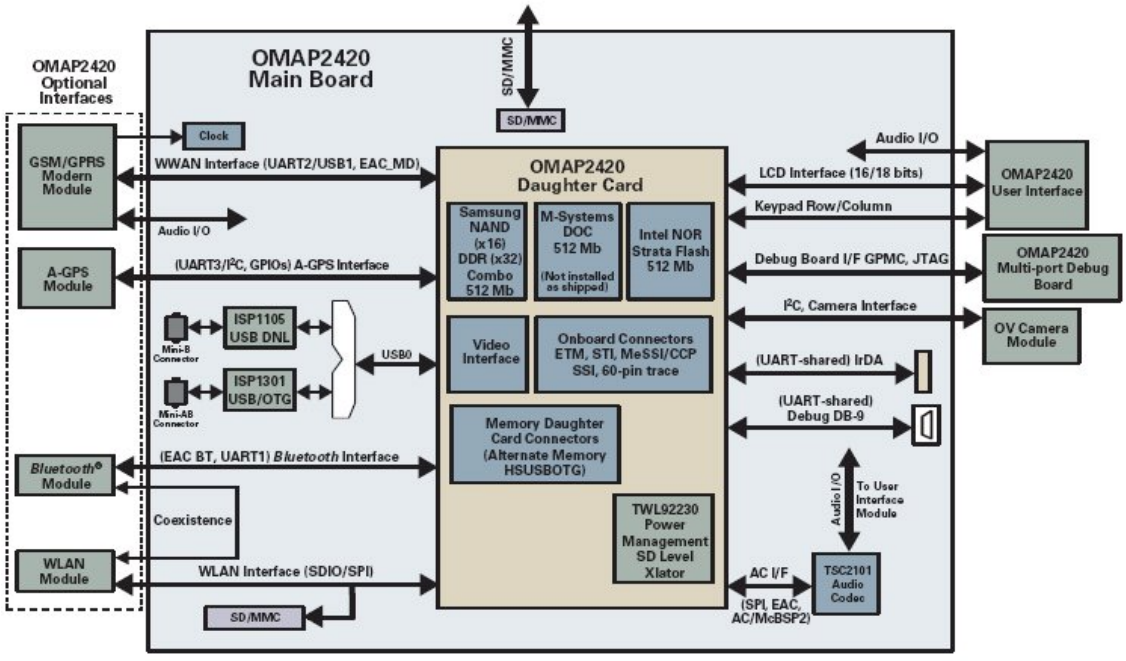


Figure 2 Board Support Package of Texas Instruments OMAP 2420 Application Processor [TI, 2005]

Embedded system development often starts with the board support package, later the development moves onto the prototypes with the product design ready implemented. Frequently, the early prototypes are produced with special hardware peripherals that are

used to connect to tracing or debugging devices. These special hardware peripherals can be a dock, on which the prototypes are plugged, or they can be an IC board attached to the prototypes. Firmware development acts an important role in the embedded system development, especially for the high-performance intelligent products like smartphones. In the early phase development, operating system is ported to the board support package or prototype, and device drivers are implemented for various hardware peripherals. Once the overall quality of prototype reaches certain mature level, the integration of UI framework can start.

### 2.3 Platform-concentrated embedded system development

Many embedded device manufactures produce a wide selection of products, from the low end to the high end, to target on different market segments. Those products are usually built on a few main product platforms to minimize the research and development cost on each product, and reuse the design and implementation among the products on the same platform. A group of products having the similar functionality are often based on the same platform. Once the first product has been developed on that platform (it's often called the lead product), common code and design can be reused on the consequent products. Furthermore, the platform evolution is frequently continuous and incremental, meaning that the new platform has some new technology integrated or has updated a few components but inherits the majority of the old platform. This kind of platform-concentrated development has the advantages of more cost-effective R&D and shorter time-to-market, over the product-concentrated development.

The development of an embedded system has gradually become a joint cooperation between device manufactures, silicon chip vendors, design firms, and software firms. As an example explaining the procedure of developing a smartphone, the SoC, flash memory ICs, and other hardware components can be contracted from several silicon device vendors; the OS can be licensed from a software firm; the UI framework can be licensed from another software firm; the product design can be out-sourced to a design firm; and the testing of final product can be out-sourced to a software / hardware testing-specialised firm. The core developing team can include only the engineers who develop hardware-dependent software such as device drivers, and the engineers who integrate software in different layers together to work out a system solution. The importance of this cooperation has been amplified by the fact that modern technologies have been evolutionary developing, and not a single company can follow the developing steps in an entire production field, but rather master the expertise in a relatively limited field. In CTIA WIRELESS 2006 conference, Nokia Chairman and CEO Jorma Ollila concluded that the success of the company was lying on the fact - "innovation not only in mobile devices and services, but also in our collaborative strengths" [Ollila, 2006]. His speech implies the cooperation has been well conducted

both inside Nokia, and between other companies in the information and telecommunication industry.

Due to the close cooperation to develop embedded systems, testing and verification become vital to guarantee the product quality and to keep the production schedule. Each component should be carefully tested and provide the exact required functions as designed before integrating it to the platform. After the integration, testing on the platform should be carried out to verify it against the desired feature. Only after component level and platform level testing and verification unveil a good result, the product development can start. If testing and verification are only performed on the final product, and an error is detected, it is difficult to locate the cause of the error. Besides the above reason, it is expensive and time consuming to fix the error on any component and then redo the components integration on the platform level. Therefore, careful testing and verification is vital to the success of an embedded system.

# 3. Basics of Symbian OS

Symbian OS was developed based on Psion Software's EPOC Release 5. The Ericsson R380 published in year 2000 was the first smartphone developed on EPOC Release 5, on which external software applications cannot be installed. Starting from the version 6, Symbian replaced EPOC as the name of the operating system. Symbian OS v6 has become an open platform. Software applications can be installed to the OS and UI framework can be integrated on the top of the OS. Nokia 9210 communicator has chosen Symbian OS v6 as the operating system. In the 9210 communicator, Nokia has its series 80 UI framework integrated with the Symbian OS [Wikipedia - Symbian OS, 2006].

Symbian OS competes with other smartphone operating systems, such as Windows CE powered Windows Mobile, Palm OS and Linux. The advantages of Symbian OS over others are debatable. Device manufacturers and cellular network operators could choose any alternative depending on their needs. In 2005 Symbian had slight over half of the market share, followed by Linux, then Microsoft's Windows Mobile. Linux experienced a sizeable gain in shipments during the second half of 2005. [Allen, 2005] In February 2006, Symbian announced that it would cut half of the licensee fee to "encroaching competition from Windows Mobile and Linux". [Symbian, 2006]

## 3.1 Smartphone overview

Smartphone is the kind of device that combines the functionality of the mobile phone and PDA. Smartphone offers the personal data management functions, such as calendar, task list, e-mail access and camera beyond the voice service. One of the key features of smartphone is that software applications can be installed to the smartphone. These applications can be developed by device manufacturers, by operators or by any third party software developers. It's a loose term to define smartphone as to have both the functionality of PDA and phone. Many recent mobile phones support the basic PDA functions like the calendar and task list, but they are not generally considered as smartphone. The OS of the handset becomes one identifier for whether the handset is a smartphone or not. [Wikipedia-Smartphone, 2006]

Smartphones are in the frontier of the technology appliance and software integration. Device manufacturers are willing to apply the latest technologies, developed for multimedia broadcast and wireless connection, to smartphones. Some of these technologies like MobileTV and WiFi have been implemented to smartphones. Tests have been carried on jointly by device manufacturers and operators. Many common office software applications are integrated to smartphones, which makes the operations like reading PDF file and editing presentation slides also available in smartphones.

Many research firms have predicted that the market share of smartphone would expend in a few years, as the increasing demand for data communication applications and multimedia applications. According to [Canalys, 2005], the global shipment of smartphones has 75% year-to-year growth in the third quarter of 2005.

### 3.2 Characteristics of smartphone OS

The concept of operating system (OS) is briefly introduced here to present readers the prerequisite knowledge for understanding the characteristics of smartphone OS. The operating system is the software managing the overall operation of a computer. It is the first loaded software when computer boots up, the OS further loads the device drivers and other necessary software to completely boot the computer to normal operation mode. The OS manages integrating new hardware to the computer, and loading application software. The OS also manages disk access, memory allocation, tasks scheduling, and interfacing with users. The most common smartphone OS are Symbian, Linux and Windows Mobile. Besides, Palm OS developed by PalmSource and BREW developed by Qualcomm also share the market of the smartphone OS.

Smartphones have special requirements for the operating system. A smartphone, like other embedded system, is supposed to be running reliably for a considerably long time. Users might not reboot smartphones frequently compared to PCs. Important user data such as contacts, calendar entries, and saved files need to be conserved safely. Fundamental features like telephony and messaging should function properly. These require the OS to work reliably in exceptional situations, for example running out of battery, losing network connection and application software failures.

The OS reliability has impact on the maintenance. There are hundreds of thousands of mobile devices shipped to market each year. The device manufacturers cannot afford to distribute a service pack to update the OS on every shipped device if any OS function failure has been detected. The OS needs to be up and running reliably in months or even years without frequent reboot, and without service packs update.

The smartphone OS needs to be responsive. It is considered to be an unfavourable user experience if a smartphone responses user input (for example, key pressing) sluggishly. The performance and the cost need to be well balanced on a smartphone. Providing a fast processor and high memory capacity, OS can give relatively good performance, but it increases the cost of manufacture. The ideal smartphone OS needs to provide an efficient performance based on the constrained hardware resources. The smartphone OS should be able to prioritize different tasks and do the scheduling according to the available limited resources to give the best performance.

The smartphone OS needs to have good extendibility and adaptability. It should support the latest technology integrated, such as the wireless LAN. It should also easily be adapted to different hardware platforms, and should support customized UI

framework by different device vendors. With a well-modelled smartphone OS the development cycle of a product, other than the lead product on the same platform, can be very much shortened. Integration becomes the major part of the development cycle, once after the lead product has been successfully developed on a product platform.

### 3.3 Symbian OS design

Symbian OS and its kernel - Epoc Kernel Architecture 2 (EKA2) - are modular. Many operations are based on the client-server architecture. For example, disk operations are performed by the file server, messaging operations are performed by the messaging server, and screen and user input operations by the windows server. The fundamental element of the OS is the kernel EKA2, which is responsible for memory management and task management.

Symbian OS is priority-based multi-tasking OS, as the kernel switches the CPU time between multiple tasks (multiple threads). The kernel does not wait for any thread to relinquish the CPU time to make a content switch, rather reschedule the tasks based on task priorities. The kernel implements the priority inheritance. If high-priority threads are waiting for any mutex held by a low-priority thread, the kernel will assign the low-priority thread the maximum priority of the threads in the waiting list. This operation minimizes the delays to high-priority tasks. [Sales, 2005]

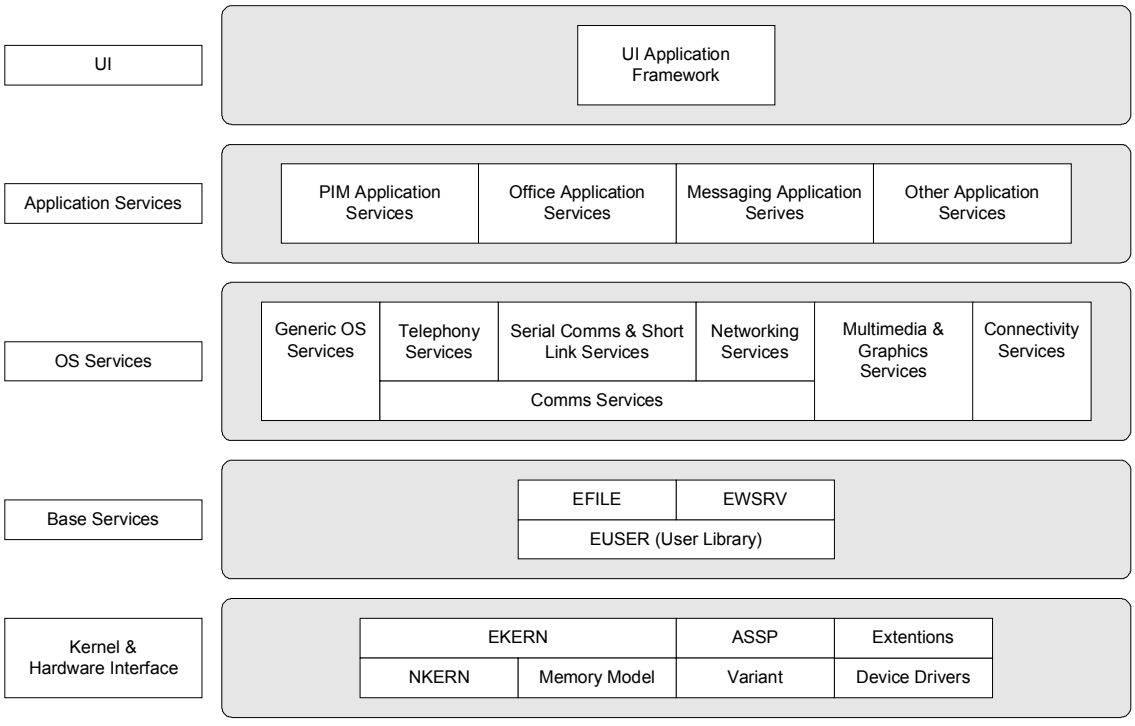


Figure 3 Symbian OS Architecture [Symbian OS Essential, 2005]

Figure 3 depicts the architecture of the Symbian OS v9.1. At the bottom is the thin kernel EKA2 with hardware interface, other layers above the kernel belong to user-side (or user-mode). The nanokernel (NKERN) provides simple, supervisor-mode threads

along with the most basic scheduling, synchronization and timing services for Symbian OS kernel. The nanokernel provides the Real Time Operating System (RTOS) features, which guarantees the kernel operations are deterministic or predictable but not necessarily fast. The memory model encapsulates low-level memory operations, such as memory mapping and doing context switch in cooperation with scheduler. The Symbian OS kernel (EKERN) provides kernel services, which include creating user-mode processes and threads, loading the Dynamically Loaded Libraries (DLLs), performing inter-process communication, and other kinds of kernel services. [Symbian OS Essential, 2005]

Smartphones typically have the microprocessor unit (MCU) and other processing units, for example digital signal processor (DSP), imaging and video accelerator (IVA), and display subsystem integrated into one semiconductor device, which is commonly referred as SoC (System-on-Chip). This device is also known by other two names – ASIC (Application-Specific Integrated Circuit) or ASSP (Application-Specific Standard Product). ASIC is customized for a particular use, ASSP is rather intended for general commercial use, but all the three terms are used imprecisely and interchangeably. All the other hardware peripherals outside the ASSP are referred as variant. On the Symbian OS kernel layer the ASSP and variant are kernel extensions, which encapsulate the hardware dependent software services. These services include timer, interrupt handler, and power management that are frequently used by the Symbian kernel. Device drivers provide the interface between the hardware peripherals and the Symbian OS. They are also kernel extensions lying on the Symbian kernel layer. [Symbian OS Essential, 2005]

One layer above the Symbian kernel is the base services layer. The user library, EUSER, is the interface between the Symbian kernel mode and the Symbian user mode. All the Symbian user mode threads gain access to kernel services via the user library. The user library has been well maintained while the Symbian kernel develops from EKA1 to EKA2, to minimize the modification to the user mode applications caused by the kernel development. The file server, EFILE, provides functions to user mode threads to manipulate the directories and files. The window server, EWSRV, provides functions to user mode threads to access the screen and keyboard. [Symbian OS Essential, 2005]

The OS services layer is also referred to as the Symbian middleware. This layer provides the majority of the OS services to users and developers who implement the user mode applications. As Figure 3 indicates, various services can be found in this layer, and it has become the richest layer in the OS from the feature point of view. The layer above is the application services layer, which is also referred to as the Symbian application engines. The OS built-in applications such as Contacts, Agenda and Browsing are implemented on this layer. [Symbian OS Essential, 2005]

The layers up from the bottom till the application services layer, all together are referred to as the Core Symbian OS, or Generic Technology. Beyond the Core Symbian

OS is the UI application framework layer. Different smartphones can have different UI designs. These UI designs are called UI application frameworks. Device manufactures license the Core Symbian OS and have an option to either implement their own UI designs, or license a UI design with certain customized modification. As all the Symbian smartphones have the common Core Symbian OS, they are compatible with each other from the third party software developer's point of view.

# 4. Debug Symbian Applications with Emulator

An emulator is the implementation of Symbian OS, instead of porting onto hardware it resides on the hosting platforms such as Windows or Linux. An emulator can be launched as a process within the hosting platform. Emulators are the most common tools for developing Symbian OS applications. During the smartphone production, emulators are often used to develop and debug the applications beyond the Core Symbian OS layer. For example, the development of the S60 UI application framework is very much based on the emulator environment. The third party software development firms use emulator to develop Symbian OS applications, which are shipped either together with the smartphone sales package, or as a separate software package that can be installed to smartphones by end users.

The principle goal to design the emulator is to support development and debugging using standard IDEs (Integrated Development Environment) on the host platform, and provide as close emulation as possible of Symbian OS running on the target hardware. Symbian software often relies on the OS client-server architecture and has frequent access to Symbian kernel services. This requires the emulator to provide the identical kernel services and scheduling as those on the target hardware. The ideal solution is to port the complete Symbian OS to the host platform. Figure 4 compares the architecture of the Symbian OS ported to the ARM processor and Symbian OS ported to the host Windows OS. Except the lower layer having the emulator-specific nanokernel and memory model, the Symbian Kernel and upper layers of the emulator are identical to the target hardware. [Sales, 2005]

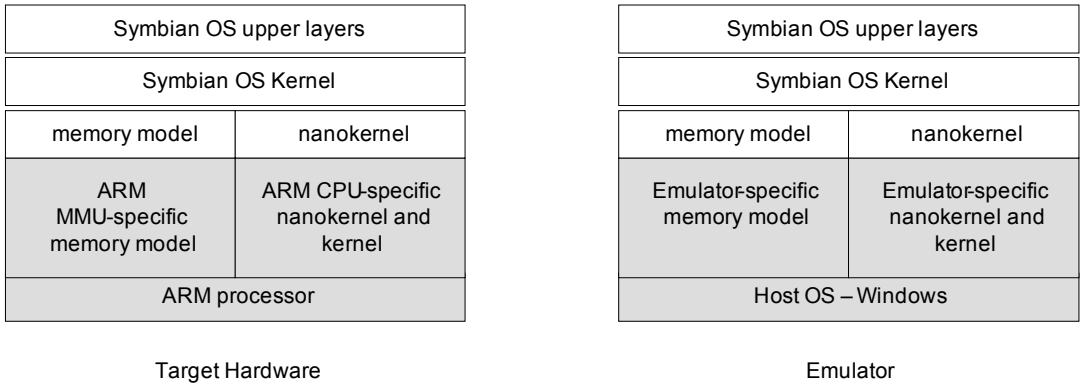


Figure 4 Target Hardware v.s. Emulator

Using emulator to debug has the advantage of easy installation and integration. Many popular IDEs support the emulator integration as a plug-in component and developers can use IDEs' debugging facilities to set a break point, single step in or step out the running thread, and monitor the memory allocation and threads life cycle. Some IDEs have also provided the functions to create the application's UI framework, and leave the developers to code and debug only the application logic. Some IDEs have the support for multiple device UI interface layouts, so that developers are able to launch

the application with different device UI layouts and test the portability of the application over several popular device UI layouts on market. Well-known IDEs supporting Symbian OS emulator integration include Microsoft's Visual Studio, Metrowerk's CodeWarrior and the coming Nokia's Carbide that is based on the open source framework Eclipse.

Using emulator is usually free of charge, and many online Symbian development communities are available for developers. Device manufactures provide emulators on their online developer support sites, and often organise developer conferences to give technical training. Developer communities provide industries news, developing tools update, and developing ticks. Some of them also offer paid professional technical support, which help developers of different levels on their work.

The emulator is quite often used on the stage when all the OS servers have been well implemented and are able to provide the proper services to applications. On this stage, the product should reach certain mature level or be already on market. With only emulators, not every application can be completely implemented. Some applications need to use the communication channel such as Bluetooth or need to access the GSM network to send or receive messages. To address these issues, either the application needs to be tested on the target hardware, or the emulator needs to be configured to replace the required services with substitutes on the host operating system. For example, if the application requires the Bluetooth communication, the emulator can be configured to use PC's Bluetooth hardware as a substitute. If the application requires sending or receiving messages, an 'Inbox' folder and a 'Send' folder can be created on Windows OS file system and the emulator can be configured to read messages from the 'Inbox' and write message to the 'Send'.

Emulator itself is an application running on the host operating system. It requires frequent updates for bug-fix, and to support the latest Symbian OS release. In any case, emulator cannot replace the target hardware for application development. Certain issues like timing the thread, object life cycle, and memory allocation on emulators cannot behave identical as on target hardware. Developers are always recommended to carefully read the release note of the emulator (usually the behaviour difference between the emulator and the target hardware are written in the release note) and be aware of the difference. Applications developed on Symbian OS need to be completely tested on target hardware. The procedure of first debugging on emulator then testing on target is common to embedded system application development.

The embedded system development has the characteristic of first releasing the product platform, and then on top of which various products get developed (these have been discussed in Chapter 2). For most of the device vendors, the emulator release follows the platform release, and usually one emulator is developed for each platform release. This also reflects the fact that developers wish the application developed on one

platform to work on all products based on that platform. In practice this rule is often tolerated. Although two products are based on the same platform, certain malfunctioned code can behave more vital and severe on one product than on the other. This causes that the problem application might work on the emulator and on one target hardware, but fails on the other target hardware. It is often difficult to solve such a problem, and requires repeatedly debugging and testing on the emulator and target hardware. Sometimes other debugging tools then emulators are needed to solve the problem.

Metrowerk has developed an on-device-debugging agent – MetroTRK, which is similar to emulators from the perspective that they both integrate into an IDE and utilize the IDE's debugging facilities. MetroTRK can only be integrated into Metrowerk's CodeWarrior IDE. It requires a similar Symbian environment on the host operating system as the emulator does. It installs a debugging agent application to the target hardware to establish a communication channel (can be USB or Bluetooth) with the IDE and pass commands and parameters between them. When debugging an application, a copy of executables and resources are first transferred to the target hardware. Developers can then use the debugging facilities provided by the IDE to control and monitor the code execution. The desired debugging operations from the IDE side are transferred to the agent application who issues interrupts to Symbian kernel to control the code execution. On the other communication direction, the agent application gives information of the executing units to the IDE who then presents these data to developers. If any library is loaded during the execution, the IDE is informed to load the same library from the Symbian environment configured on the host operating system. The MetroTRK gives more real time application execution information compared to the emulator and is able to debug on the target hardware.

To summarise this chapter, the advantage of using emulators is that they can be integrated to IDEs and utilize IDEs' debugging facilities on Symbian application development. As most application developers who have worked on the Windows platform are familiar with IDE's debugging functions, using emulators have shortened the learning curve for them to start doing Symbian development. Maintaining and updating emulators do not require any special knowledge or hardware, and many developing resources can be easily accessed from online developer communities or manufactures' technical support sites. These features make emulators the most popular debugging tool among developers. What we need to bear in mind is that emulators cannot simulate application running identically to target hardware, and applications have to be completely tested on target hardware before they can be released.

## 5. Trace Symbian Application Execution

Tracing is defined to monitor important variables while following application executing path, and check code branches coverage. It can be represented as a way to collect all kinds of data that developers wish to investigate while executing the application. Tracing is an efficient way to debug applications, because it collects real time data without setting any break point to pause application execution. Typical tracing needs several steps. First, code need to be instrumented at the desired places, which are often function entry point, function leaving point, and selective statements like *if / else* and *switch*. Then the instrumented code is compiled and application executable is built. The last step is to execute the application and collect the trace outputs while running on the target hardware.

Tracing has been used widely in various application developments, and the procedure of tracing has been kept unchanged. This procedure can be summarized in four phrases, which are instrument code (add traces), compile code, execute code, and collect trace. For example, many developers started with the familiar "Hello World" application, where the idea of tracing is used. The collected trace output (the printed string variable "Hello World") indicates the application has been executing properly. When tracing is used in application development, various functions and variables can be traced. Developers compare the collected trace output with the instrumented codes to see if the application executes as desired.

Tracing application execution on Windows platform do not need any extra hardware, the tracing output can be collected from either the application output console or the IDE's application output window. To debug applications with tracing, usually every function's entry points and leaving points are first instrumented. The common way to instrument code is to simply place a print statement that outputs a string to indicate an event happens, for example, entering function event or leaving function event. When these trace outputs have been collected, developers can check the application execution path. If application does not execute as desired or it exits with an error in the middle of a function, developers can instrument the codes to add more print statements to the functions that are suspected places of the error. Then recompile the code, execute the code, and collect the trace outputs to make a further investigation. Debugging application with trace is an incremental procedure. Every time after traces have been collected, developers may need to adjust the previous traces or add more to better uncover application defects. This procedure enables developers to gradually understand how the code is executing and to fix bugs.

Tracing application execution on embedded systems often requires special hardware and software to collect traces, interpret them to understandable messages, and display these messages. This is due to embedded systems usually do not have enough process capacity to execute the complete tracing procedure, and embedded systems

often have no display device or too small display device to present the tracing outputs. The solution for these issues is to output the traces in the form of unformatted binary to a defined channel, which is monitored by a customized hardware – the trace box. Then the trace box either formats a single trace to a message, or composes multiple traces and formats to a message. The formatted messages are routed to a PC, where they are decoded and results are presented to developers. (See Figure 5)

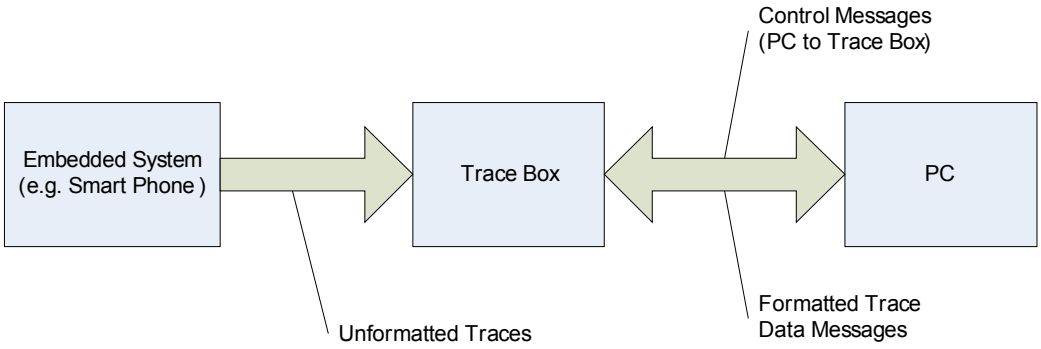


Figure 5 Trace Embedded System

Tracing can be used vertically in every phase of the embedded system development, from the low level firmware that is closely coupled with hardware, through the operating system middleware, to the high level UI applications that reside above all other layers. Debugging embedded system to locate and fix software defects usually requires repeating the tracing procedure. When applying to firmware development, tracing repeatedly may take longer time compared to Windows development. This is because when traces have been altered, the entire firmware may need to be first rebuilt and reloaded to the target device, and then reboots the target device to collect the instrumented traces. Chapter 2 has explained that the firmware often reside on a flash memory IC or a programmable IC, and any update to it requires either reload the complete firmware to flash memory or reprogram the IC. In practice it implies that firmware development requires much more careful design and implementation as rebuilding firmware is time consuming while debugging.

For embedded system development, tracing has a performance impact on application execution. The performance of the application, which outputs many string variable traces, is slowed down because of processing string variable often takes quite much CPU time. The impact can be sometimes ignored if the application execution is not time critical. A more efficient way to do tracing on embedded system is to encode common traces into integers. For example, instead of outputting a string variable while entering certain function, an integer variable that encodes the function entry point can be printed. Tracing with integer variables has minimum impact on application performance but requires extra work to encode information to integers when compiling applications, and later when traces are collected the identical information need to be decoded from those integer variables.

A similar debugging method to tracing is to write a log file, which is often used on debugging the server side applications. Because servers are supposed to be running several days (even weeks or months) without interruption, the usual way of debugging, meaning set a break point and single step in / out application execution would cause the service break and other inconveniences. Often defects on server side applications cannot be easily reproduced and they are usually triggered by some events. Until the trigger event is known and the defect is reproduced, the server side applications can hardly be debugged. Writing a log file is similar to trace application execution in the way that trace outputs are collected in a log file. Developers can then locate the defect and know the event causing the defect by investigating the log file.

To summarise this chapter, trace application execution is a common debug method that is used in various application developments. The fundamental part of tracing is simple and kept unchanged, meaning instrument codes, compile codes, built application, and collect traces. Debug application with tracing is time consuming, as the procedure is repeated while gradually understanding application execution. Often when there is no other fast way to debug, tracing is used instead but it remains as an effective way. With the maximum use of tracing, meaning print out everything along with the application execution, eventually bugs can be fixed. For embedded system development tracing needs extra hardware and software, and is often used in the early phase of smart phone development when there is no emulator available.

# 6. Solve Symbian OS Software Failures with Mobile Crash

In practical tracing is used most to implement firmware at the very beginning of the product development life cycle, for example device drivers implementation, although tracing can be applied to any development phase. On the contrary, emulator is used when products reach certain mature level, and mainly targets on Symbian application development or Symbian OS middleware development. They both are used in the R&D environment, meaning during the period of product implementation. Following this phase there is a vast period of testing the product and maintaining the product.

Even the smartphone has been completely tested in the laboratory, there can still be unfound bugs, which may need to be uncovered by performing a sequence of events. To reduce the potential error count and to reach a relatively high quality before releasing the product to market, it's necessary to form a group of end users to test the smartphone in certain period. In this kind of testing, the end users are told to use the smartphone as their personal cell phone and use it as much as possible. The testing very closely simulates how customers use the smartphone in their daily life, and it usually uncovers bugs that are difficult to be found in laboratory environment.

The end users are often unable to record the exact steps before certain software component fails or panics. Consequently, it makes difficult for developers to reproduce and fix the bug. Sometimes it needs lots of communication between the end users and developers to try to reproduce the failure situation, and fixing one bug requires much time and efforts from both sides. Hence, the smartphones under testing need to have a debugging utility to capture software failures, and to collect information of the panicked component for the purpose that the bug can be located and fixed by analyzing the collected data. This idea becomes the very first requirement of developing Mobile Crash, which could automate the procedure to capture, transfer, and analyze the Symbian OS software panics. (See Figure 6)

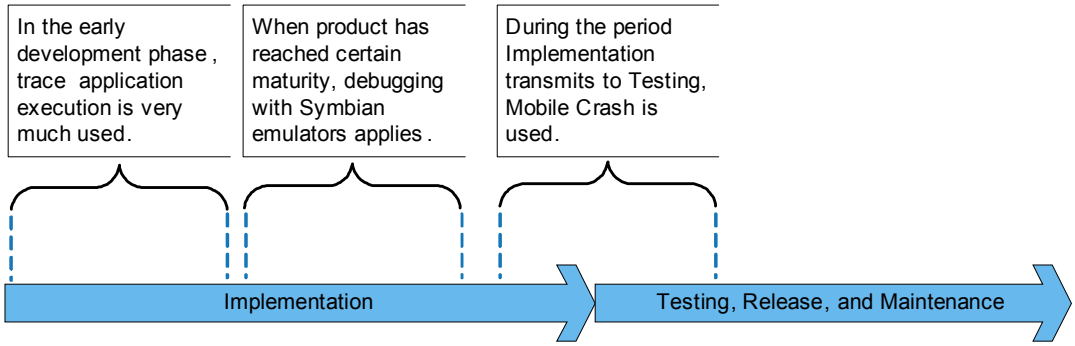


Figure 6 Debugging Symbian applications in different development phases

## 6.1 Analysis of developing requirements for Mobile Crash

Realising there is a need for such a debugging utility that would be used in the transmission period from development to testing; we must consider the needs of both

testers and developers. From the tester point of view the module should be easy to use, highly automatic, and keep the tester notified what happens in the background operation. From the developer point of view the module should be able to collect real time panic data, keep data lost low, and validate data at each step during the transmission. Besides, the module needs to have less dependency on the hardware so that it can be ported to other smartphone platforms. Also if the panic data is transferred through the GSM network, the data load should be minimized in order to reduce the networking cost.

More detailed requirements need to be specified on different perspectives of the project. These include software implementation, hardware dependency, installation method, and customer usability. Among these the software implementation remains as the key issue, and it affects the other perspectives. This is because the different ways applied to the software implementation, for example, loading libraries of common Symbian OS or libraries of a certain specific platform affects the project compatibility among different hardware, and further affects installation to different products. For the software implementation, the core is to develop a mechanism to capture Symbian panics. This mechanism should work among all the products based on the Symbian OS, and should capture both the Symbian kernel side faults and Symbian user side panics. It should further collect sufficient amount of data to analyse the panics. If the software core has been designed and implemented properly, other perspectives of the project requirements will be met consequently or with an effort of moderate modification.

To design the core, first the necessary data for analysing panics must be defined. Developers need at least have the information of the panicked software module, panic category and panic ID to know which component has failed (The relevant knowledge of Symbian OS panics is discussed later in Section 6.3). But these data are not enough to locate the bug on code lines or give any hint to fix the bug. More relevant data that could be used to debug software failures are the panicked thread call stack and loaded binary modules at the time the panic happens. With these data, the function pointers pushed onto the stack can be decoded. Developers are then able to analyse and work out the function calling sequence before the panic, and further pinpoint suspicious code. Table 1 defines all the data that are required to be collected in the case a Symbian OS panic would happen.

| Data Item | Definition |
|---|---|
| Timestamp | The timestamp when panic happens |
| Panicked Module | The panicked software component/module |
| Panicked Process | The process being panicked |
| Panic Category | Symbian OS defined panic category |
| Panic ID | Symbian OS defined panic ID |
| ROM ID | Rom image identity |
| SW Info | The smartphone SW information |
| Language | The smartphone language |
| IMEI | The smartphone International Mobile Equipment Identity |
| Program Counter | A CPU register holds the instruction being executed |
| Stack Pointer | A CPU register used to access call stack and points to the current top of call stack |
| Stack Base | The bottom address of call stack |
| Call Stack | A stack stores information of the active program execution |
| Loaded DLLs | The lists of dynamical linked libraries loaded to memory |
| CPU Registers | The CPU register values |
| Reset Reason | The software reset reason if the panic causes system reboots |
| Test Set | Developer defined test set that causes the panic when executing the test |

Table 1 The Mobile Crash collected data when Symbian OS panic happens

Among the items listed in above table, some can be used to debug the software failures, and others can be used to evaluate the product maturity. IMEI code can uniquely identify the smartphone. Software info can be checked to evaluate whether the smartphone firmware is reliable to be used on the official product release. Panicked module uncovers whether certain software module is robust when integrated to the OS. Timestamp can be used in the way that defining an evaluation period to evaluate firmware or certain software module quality.

All the collected data need to be stored in the smartphone storage media with a well defined format, so that when they are transferred to developers for analysis, they can be decoded according to the format. Defining a data format also benefits validating data during the transmission. For example, for each captured panic the format may

specify the number of collected items at the beginning of the data chunk, and append a CRC (Cyclic Redundancy Check) value at the end of the data chunk. The format may also define to apply character encoding to textual data items like SW Info and Panic Category, and store other data items like Call Stack and CPU registers in binary format (a sequence of bytes). When these are specified, reading and writing data can be standardized to refer to the format, and data can be verified against the format as well.

At the very first chapter of this thesis, the overview of Mobile Crash described that the captured panic data can be transferred to the decoding server either by SMS through the GSM network, or by USB first to a workstation then route the data to the server via corporate LAN. From the perspective of testers, the requirement for data transmission is to automate the procedure as much as possible. The reason behind are testers may not necessary stay in a lab environment, or have the access to the corporate network when panic happens. So transmission requires another way than relaying on the corporate LAN, and it needs to be accessible at any time. For simulating real life smartphone uses, testers are usually selected randomly from a volunteer group and may not have the required knowledge to manipulate a moderate complex data transmitting operation. Further more, the product under testing often has to follow a tight schedule to complete the testing procedure and minimize the delay to release to market. These facts combined decide that the method used in the data transmission should be automatic and immediate after panic is trapped. Panic data need to be transferred for debugging and analyzing without much delay. Transferring panic data via SMS fulfills the requirement explained above.

Considering the data load one SMS message can bear is fairly small, the captured panic data need to either be compressed or part of the less important data to be removed. Even though those methods are applied, one SMS can barely carry all the data belonging to one captured panic. This requires a concatenation protocol being designed for both the sending and receiving components. It works so that the captured data of one panic can be split into several parts and each is sent by one SMS message, and these messages are concatenated together when received to reconstruct the original data chunk. In addition, the smartphone user need to be notified when panics are captured and SMS messages are sent, so they will not wonder why there are many SMS being sent from the phone.

However the amount of data load does not affect the USB transmission, which is limited to be used inside the corporate LAN. Panic data transmitted in this form is eventually routed to the decoding server through the corporate LAN. Yet Mobile Crash is not limited only for testers, developers can use it along to debug Symbian OS panics. In the later case, developers should have the decoding software installed to their workstations to decode the panic data locally, and the USB connection becomes the preferred data transmission step.

Nevertheless, the panic data can be transferred by other ways than SMS and USB. For example, the panic data can be written to the memory card of the smartphone, then read the data from the card. The general guidelines of data transmission are that, if Mobile Crash is to be used during the testing period, the panic data are routed to the remote decoding server for decoding, and more importantly for collecting data and calculating statistics to evaluate the product maturity. Otherwise if Mobile Crash is to be applied to debug application, the data can be transferred to a local workstation to get decoded, provided that the decoding software is installed to that workstation.

After the requirements have been specified for capturing and transmitting panics, the requirement for decoding panics is discussed here. Panics need to be decoded regardless what underneath product has generated them, as far as the product is developed based on Symbian OS. Product platforms differentiate each other at the hardware layer, which consequently affects porting Symbian OS to the platform and memory layout for the core Symbian OS libraries. The decoding application should be configured to decode the panic according to the original platform that raises the panic. It is especially meant for decoding the dumped call stack, as the call stack stores merely the function return addresses that are mapped to different binary modules depending on products. The decoding application should be able to map the call stack symbols that are of binary format to function names of text format.

In addition to the decoding application, Mobile Crash decoding server needs to have the database set up to store panic information after panics have been decoded successfully. Statistics can be calculated by making queries to database. For browsing the panic info and viewing the statistic graphics, the web service is held in the same server. The provided services are closely tied with the database. End users could issue a request to the web server, which will process the request by mining the database and collecting the required data, then the data is presented or statistic graphics are drawn.

When the SMS message is used to transmit the panic data, the server should have a message gateway application stand by running. The gateway application is installed to receive SMS, concatenate panic data if the data dump belonging to one panic has been split and loaded to several messages to send, and feed the received panic data to the decoding application. The gateway application should concatenate messages according to the same protocol that is used to split the panic data. A time-out mechanism needs to be designed so that if one SMS from the concatenated chain is lost, the rest of SMS messages from the same chain are discarded after certain time-out period. The received panic data needs to be verified before redirect to the decoding application, which guarantees data has not been altered when transmitting through the GSM network.

## 6.2 The study of Symbian OS panics

This section gives a brief survey of the Symbian OS panic. There are two main groups of panics defined, which are the Symbian user panic and the Symbian kernel panic. These are defined to identify where the panic is raised, either from a user side thread or a kernel side thread. The concepts of user side and kernel side are varied by different privilege boundaries. A user side thread does not have access to any memory space outside the scope of the owning process, on the contrary, a thread reside in the Symbian OS kernel can access any memory space of the OS. Usually if a user side thread panics, the thread itself gets killed, and the application process that owns the panicked thread is closed. If a kernel side thread panics, it's expected that the kernel can not function properly afterward. Hence a Symbian OS reboots usually follows.

Panics are errors caused by careless programming. Typically panics are raised by passing illegal parameters to library functions, which are in DLLs that are loaded and running in the same thread as the application program. Then library functions call *User::Panic()* to panic the thread. Some errors may cause the Symbian kernel itself to terminate, those are often referred to as kernel fault rather than kernel panic. When a Symbian OS panic happens, the panic category and the panic ID are generated along with the thread termination. This category-ID pair specifies the circumstance that may cause the panic. Usually this doesn't give much information except telling what operation on what API function may raise such a panic. It leaves developer to check all similar API function calls used in the application program to uncover the cause of panic. Such a checking procedure might be tiresome without any debugging tool (for instance tracing application execution). All the Symbian OS defined panic categories and panic IDs are documented in the Symbian developer libraries, which can be either accessed online at Symbian's developer support site or downloaded along with the development SDK.

Symbian OS has provided utilities that can be built into the device firmware to collect data other than panic category and ID when applications or system components panic. For the user thread panics, Symbian provides the API named MinKda (Minimal Kernel debug agent) that defines functions to collect the data such as call stack, loaded DLLs lists, and processor register sets. For the kernel side panics, Symbian provides the API named crash debugger that defines the similar functionalities to collect panic information. These APIs become the foundation of the Mobile Crash development.

Symbian has defined a framework to use the MinKda API to collect panic data and also provide the reference implementation called d_exc to automate panic capture procedure. The implementation of d_exc traps the user thread panics and logs various panic data for analysis. The d_exc needs to be launched within the EShell interface, which is a text shell application (the program executable is eshell.exe) provides a command prompt mostly for running testing utilities. The functionality of eshell.exe in

Symbian OS is similar to the cmd.exe in Windows OS. After the d_exc is launched, it's running in the standby mode and monitoring user thread panics.

When a panic is trapped the d_exc enters an interactive mode, asks the user whether to log the crash data or not. If the user chooses to do so, it generates two files d_exc_<thread-id>.txt and d_exc_<thread-id>.stk on the device file system c:\ drive. The thread-id in file names is the panicked thread's id. Basic information of the trapped panic is saved in the file d_exc_<thread-id>.txt. Following is an example indicates that the 'Main' thread of the process 'Debugging Demo' has panicked. The panic category and ID are logged, and developers can refer to the Symbian developer library for what might cause such a panic. The address range of the panicked thread call stack, various registers and panic time loaded DLLs list are also logged in the file. In Table 2, CPSR stands for Current Program Status Register, the 5 least-significant bits of CPSR indicates the ARM processor mode. In the example, CPSR [4:0] has the value 10000, which indicates the processor was in User mode when panic occurred. [Symbian v9.2, 2006]

THREAD NAME:

Debugging Demo[00000000]0001::Main

PANIC CATEGORY & ID:

E32USER-CBase: 40

CALL STACK:

00403000-00405000

REGISTERS:

PC=f92c1fbc (User Register R15 has the same value as PC)

User Registers R0 – R15 (R13 is the stack pointer)

CPSR=88000010

DLLS LIST:

F92C1F28-F92C2350 Z:\sys\bin\debugging.exe

F8CD6E98-F8CD7DA4 Z:\sys\bin\eikinit.dll

F8D36708-F8D36BF4 Z:\sys\bin\techviewinit.dll

F8DA9F78-F8DAA570 Z:\sys\bin\spaneinit.dll

F8D36C78-F8D37EF8 Z:\sys\bin\Econs.dll

Table 2 Basic panic information logged by d_exc

The program counter (PC) holds the instruction being executed when panic occurs, it can be decoded from the ROM symbol files or RAM based executable map files to pinpoint the exact function that causes the panic. In the example, the program counter value falls into the address range of debugging.exe (from Table 2 the PC = f92c1fbc, the logged DLLs list points out the address range of Z:\sys\bin\debugging.exe is f92c1f28-f92c2350). The list presents where the program codes are loaded to memory at run-time. The d_exc can also trap the processor exception, in which case the panic category and id are not logged, instead more system registers are logged. These usually include R13svc, R14svc, FAR, FSR, and SPSR_svc, which are explained in Table 3. As most of the Symbian OS smartphones are constructed based on the ARM processors, understanding the processor architecture and instructions benefits understanding the Symbian panic handling procedure. But topics of ARM processor expend widely to techniques, which are out of the topic of the thesis. Table 4 lists a brief summary of the ARM processor modes and used register set of each mode. More readings about ARM processor can be found in [ARM, 2006].

| | |
|---|---|
| R13svc | Processor supervisor mode R13 indicates program counter |
| R14svc | Processor supervisor mode R14 indicates link register |
| FAR | Fault Address Register indicates the risky address that was accessed |
| FSR | Fault Status Register indicates the MMU fault |
| SPSR_svc | Saved Processor Status Register holds a copy of the CPSR when processor enters a new mode |

Table 3 Registers logged when processor exception occurs [Symbian v9.2, 2006]

| CPSR[4:0] | Mode | Used Register Set | Description |
|---|---|---|---|
| 10000 | User | PC, R14-R0, CPSR | Normal program execution mode, the program is restricted to access the protected resource. |
| 10001 | FIQ | PC, R14_fiq-R8_fiq, R7-R0, CPSR, SPSR_fiq | Fast Interrupts handling mode. |
| 10010 | IRQ | PC, R14_irq, R13_irq, R12-R0, CPSR, SPSR_irq | General-purpose Interrupts handling mode. |

| | | | |
|---|---|---|---|
| 10011 | SVC | PC, R14_svc, R13_svc, R12-R0, CPSR, SPSR_svc | Protected mode for OS. Entered after Software Interrupt (SWI) or Reset instruction. |
| 10111 | Abort | PC, R14_abt, R13_abt, R12-R0, CPSR, SPSR_abt | Exception handling mode. Entered after data abort (data access memory abort) and pre-fetch abort (Instruction access memory abort). |
| 11011 | Undef | PC, R14_und, R13_und, R12-R0, CPSR, SPSR_und | Exception handling mode. Entered after executing an undefined instruction. |
| 11111 | System | PC, R14-R0, CPSR | Use same register set as user mode, it's a privileged user mode for OS. |

Table 4 ARM processor mode and used register set [Symbian v9.2, 2006]

### 6.3 Decode Symbian OS call stack

The other file logged by d_exc is the stack file d_exc_<thread-id>.stk, which is in binary format. The file needs to be decoded to text format to be read by developers. To decode the stack file, the Core OS symbol file and RAM based executables map files are required. These files are generated by the image building tool while creating the flashable phone image file. The Core OS symbol file maps the virtual memory address range to binary modules (these may include executables, libraries and device drivers) that permanently reside on memory. Their address ranges are fixed and will remain the same unless new image file is flashed to the phone. The map files provide similar mapping, but the difference is map files are for programs that are loaded to RAM memory on demand for execution. Therefore they cannot be mapped to fixed memory address range. Because of the limited memory resource on smartphones, many modules are not placed to Core OS image, and are loaded dynamically when required by application programs. Next we first present the knowledge of Symbian OS memory usage, then the procedure of decoding call stack.

Symbian OS based smartphones use flash memory to store system code and user data. Flash memory is non-volatile, and can be electronically erased and reprogrammed. Non-volatile means that it does not require power to maintain the state of the stored data. The use of flash memory is limited by its physical construction that data are erased and reprogrammed in blocks, which means it does not support writing one byte to a random address. There are two major forms of flash memory, NOR and NAND. The principal difference between them is how data can be accessed. NOR flash is a randomly addressable memory, programming on NOR flash can operate one byte at a time. On NAND flash data can only be accessed in blocks. This makes NAND flash much like a

hard disk or memory card, as the basic date unit for reading and writing is a block. But NAND flash has faster erase time, higher density, and lower power consumption than NOR flash. And more importantly, it has a lower cost per bit than NOR flash. To reduce the production cost, smartphone manufacturers usually favour the low-priced NAND flash to the NOR flash. As saving per product would result significant profit considering the huge amount that are shipped every year.

NAND flash is a block device suitable for storage of code and data, but does not provide execute in place (XIP) of code due to its physical construction bounds that code cannot be randomly addressed and accessed on NAND flash. Code need to be first read from flash to XIP memory such as RAM, then executed by the processor. This limitation requires a complex file system operation of the Symbian OS. One solution would be to shadow the entire code area of NAND flash to RAM. However, this would raise the amount of the costly RAM consumption. Hence, more cost-effective configuration involves shadowing partial the code stored on flash at device boots up, and shadowing rest of the code on demand. Carefully choosing the code shadowed to RAM at device boots up would balance the gained speed improvement of executing code on RAM and the total RAM consumption on device.

The Core OS, which includes the Symbian kernel, kernel extensions, media drivers and file server is essential for booting the entire Symbian OS. Thus the Core OS is shadowed to RAM permanently at device boots up, and is accessed read-only via ROM file system. When building the flashable phone image, these components belonging to the Core OS are built into one single image commonly referred to as the ROM image. The rest of code on NAND flash, which is not included to the Core OS, is loaded on demand to the RAM for execution and unloaded afterward. This procedure is performed by the Read Only File System (ROFS), which interprets the virtual address of the code and shadows the code to RAM. During the phone flashable image creation, these components are built into one single image often referred to as the ROFS image. Therefore, components reside in ROFS image are RAM based executables that cannot be mapped to fixed memory address range. Rather then mounting the two file systems to separate drives, an upper level thin layer named Composite File System combines them into a single drive (the Z: drive). Depending on the request made to file server, the Composite File System passes the request to either (or both) the ROM file system or ROFS. (See Figure 7)

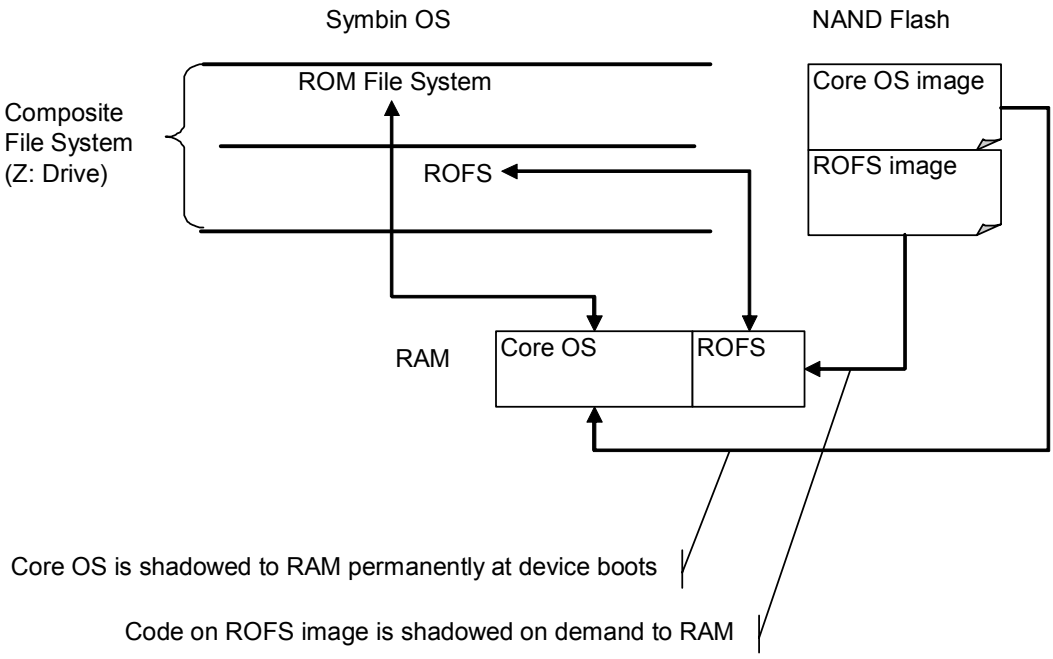


Figure 7 Symbian OS file system and memory map

To decode the call stack captured by d_exc, the ROM image symbol file and ROFS image map files are required. Components included in Core OS are shadowed to fixed addresses, and the mappings from code to address are stored in ROM image symbol file. Each program resides on ROFS has its own map file, which provides mapping from functions to address range. The mapping is made on the assumption that code is always shadowed to the same RAM memory address. But in practical code on ROFS is shadowed on demand and address may change every time shadowing happens. However the function offset to the entry address of the executable or DLL keeps the same regardless where the code is shadowed. If the stack word does not lie within the ROM image but rather within certain executable or DLL loaded from ROFS image, it can be decoded by counting offset to the entry address then comparing to the function offset value counted in the corresponding map file. (see Figure 8)

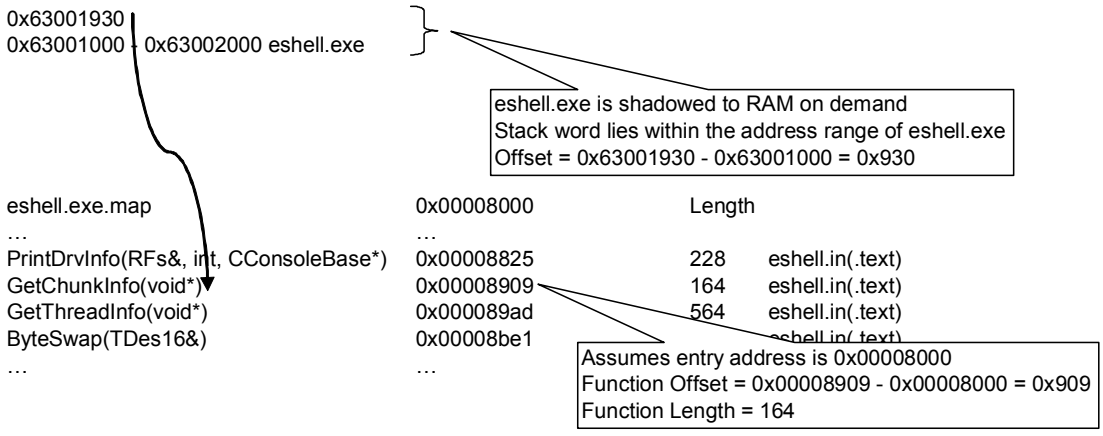


Figure 8 Decoding stack word from ROFS image map file

However, not only function return address is pushed to Symbian call stack, but also automatic variable is stored on the stack. Even when an address on the stack can be mapped to ROM range, it may point to data instead of code as there is also data in the ROM. To trace back function call sequence before the panic happens, the stack pointer value need to be retrieved first. Register R13 has the stack pointer stored, and yet different register set is used depending on the processor mode. CPSR [4:0] points out the processor mode at the time of panic, the proper R13 can be chosen after the processor mode is identified (see Table 4). For example, if panic happens on SVC mode, R13_svc has the right stack pointer.

When tracing back through the stack, a heuristic method can be used to decode the function return address. That is, assuming every stack word that can be mapped to ROM symbol file or ROFS map files is a function return address. Yet, this method cannot distinguish the return address properly in the following situations. First, the stack word that can be mapped to function may in fact points to data. Second, there may be return address from the previous function call left on the stack. An example of such situation is presented in Figure 9. Function F calls A, then B, and then C in sequence, panic happens when executing B. Function A further calls X and Y in sequence, on the stack pushes the return address of A, X and Y (Figure 9, stack snapshot 1 to 3). Note that Symbian OS stack pointer decrements when pushing new items on the stack. Once function A has returned, stack pointer moves upward to where F's return address is pushed. Then function B is called, consequently its return address is pushed on stack to where A's return address used to be (Figure 9, stack snapshot 4). At this moment panic happens, X and Y's return addresses are still on the stack although they have been successfully completed. Developers need to check the stack section above the stack pointer when tracing back the function calling sequence. In addition, if stack has overflowed, the stack pointer will have a lower address than the stack top, meaning 'beneath' the complete stack section.

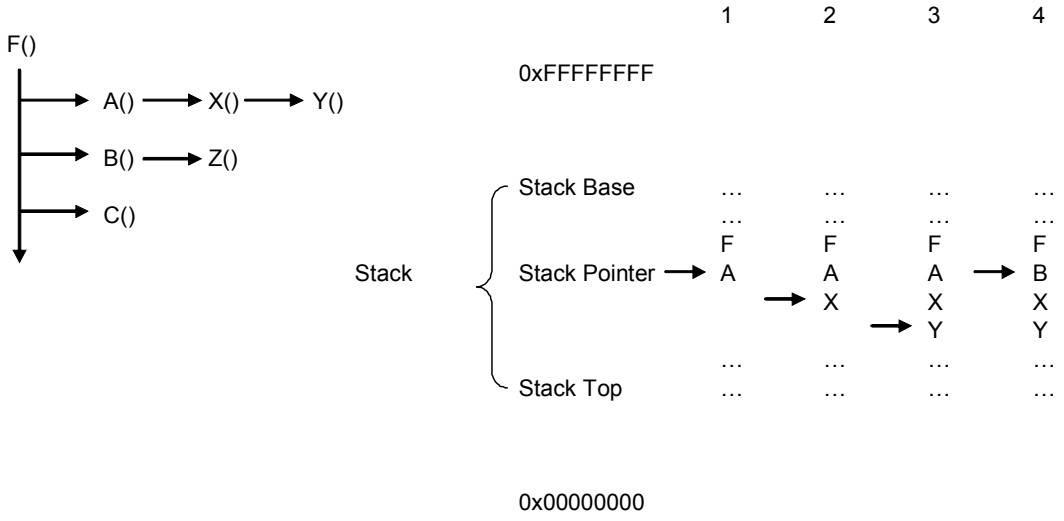


Figure 9 Symbian function calling sequence and stack

The heuristic way of decoding stack generates lots of noise, and it cannot provide precise information of pushed function return addresses on stack. It leaves developer to take a further study to filter off the noise and work out the calling sequence.

### 6.4 Mobile Crash design in general

Implementing an easy-to-use utility to automate reporting and analysing Symbian panics is essential to improve the software quality for Symbian OS based products. It becomes even vital for Symbian software development as many other factors do not favour developers to work with Symbian compared with Windows or Linux. This is partly because Symbian OS is complicated to understand as it invents new ways of doing C++ development (such as Leaving, Cleanup Stack and two-phase constructors) and strict coding frameworks to follow (for example, the Active Object framework and Device Driver framework). Partly the reasons are that Symbian API documentations are not well maintained and coding examples are relatively rare to find compared to Windows or Linux development. In addition, the developing tools are not reliable, especially lacking a handy on-device-debugging tool. Emulators often cannot identically simulate how the application is executed on target devices. Tracing application execution is precise but requires special hardware and software package that are not accessible for the majority of developers. All these urge to develop a utility that would automate software failure reporting and analysing.

Mobile Crash is implemented as a post-mortem debug utility, which means that it first traps Symbian OS panics and analyzes those panics afterward. The Symbian user side panics are handled by MinKda (Minimal Kernel debug agent), which provides API functions for collecting panic information and processor status at the time panic happens. D_EXC_MC is the component that calls the MinKda API functions and does the panic data collection. For each trapped panic, the data are further written to a crash binary file and stored at device file system. The Symbian kernel side panics are trapped by Crash Debugger, and the concerned data are collected to a dedicated memory chunk. This procedure is automatically executed by Crash Debugger when kernel faults. After kernel has panicked, no particular OS functions can be assumed to work properly, and the device needs to be reset afterward. The resetting operation is initiated by Crash Debugger after kernel fault has been handled. Crash Debugger provides API functions to check and retrieve the saved panic data from the dedicated memory chunk. At the next boots after resetting device, D_EXC_MC calls the Crash Debugger API functions to retrieve those saved panic data (if found that data exist on the memory chunk) and write them to a crash binary file. See Figure 10 for the module design of Mobile Crash.

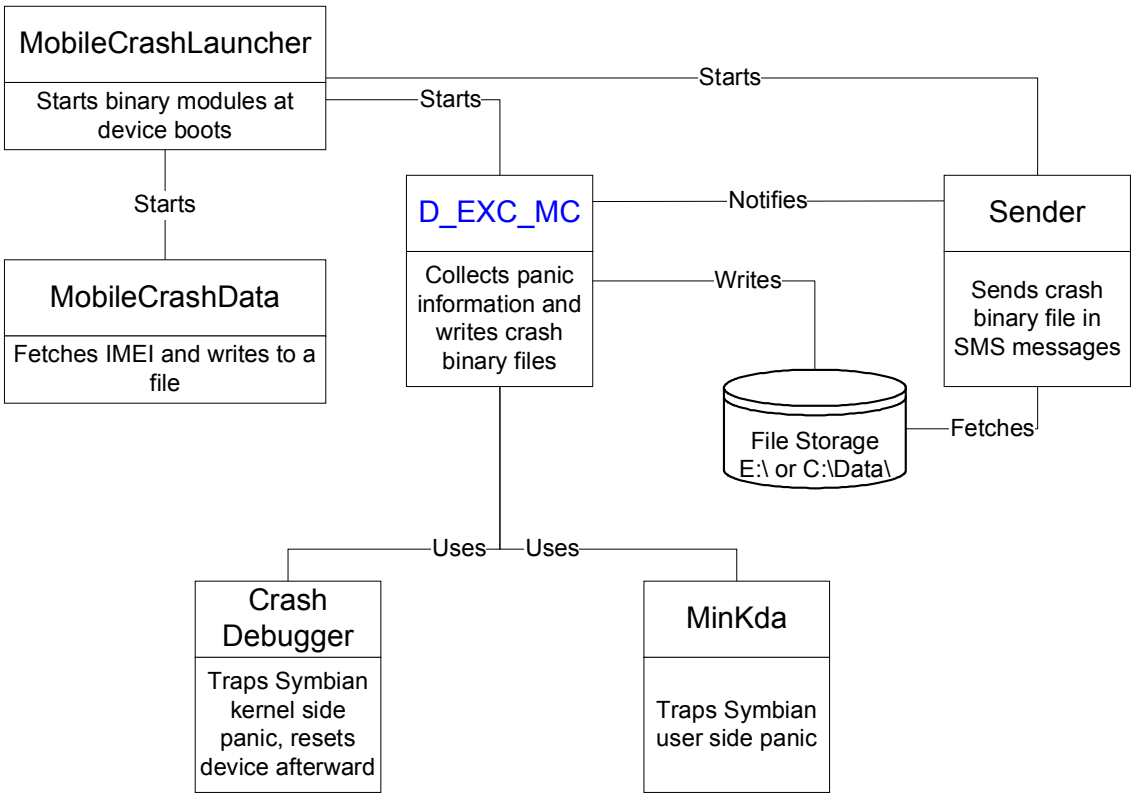


Figure 10 Mobile Crash module design

The crash binary files are transferred either via direct file copy, or via SMS messages, which are handled by Sender component. The Sender gets a notification from D_EXC_MC once a file has been written. Afterward, the Sender fetches the file to split it into several packages, and then send these packages via SMS messages. A thin binary module MobileCrashData fetches the device IMEI code and writes it to a file that is referred by D_EXC_MC when writing crash binary files. That module can be extended to gather other device specific data, such as SIM and cellular network operator information. The module MobileCrashLauncher starts the required Mobile Crash processes at the device boots, which include MobileCrashData, D_EXC_MC and Sender processes. MinKda and Crash Debugger reside on the Symbian kernel side. They are compiled to a special Symbian OS binary type – logical device driver (file extension is ldd) and are loaded by the OS during the device boots. Sections 6.5, 6.7 and 6.8 discuss logging Symbian user panic, logging Symbian kernel fault, and transferring crash binary files in detail, respectively.

The collected crash binary files are then parsed and decoded by an application named Selge.exe running on a PC workstation. Depending on the crash binary file originated device, Selge.exe loads different symbol tables to decode dumped call stack. An address pushed onto stack is mapped to software module and class member function, which is encapsulated in that software module. Data other than call stack are parsed according to their formats written in crash binary files. Selge.exe identifies and parses crash data item (other than stack) by its Item ID and Item Type, which are appended

before the actual crash data. The parsed crash information is stored to database for analyzing panics and evaluating product maturity. Section 6.9 discusses parsing and decoding crash binary files in particular.

Mobile Crash as a post-mortem debug utility is intended to assist developer to uncover software defects and locate programming error on code lines. From the decoded call stack, developers can track the function calling sequence before the panic. Further check the code of the panicked software module, and look for the functions decoded from the call stack may point out the suspicious code resulting in the panic. If any information of how to reproduce the panic is known, developers can reproduce it and meanwhile trace the suspicious code to debug the error. In practice, Mobile Crash is mostly applied to the software integration and testing phases, where software defects are detected due modules interactively work with each other and more comprehensive tests on products. The collected crash binary file together with a brief description of crash pre-condition are sent to developers to analyze and debug the error. Product maturity evaluation is drawn from the crash statistics, which are calculated based on all crash binary files collected from the concerned product.

### 6.5 Log Symbian OS user side panic

Mobile Crash handles Symbian user side panics though an executable d_exc_mc.exe. It traps Symbian user panic, collects relevant data of panic, and writes the data to file in smartphone file system. It is also possible to implement other data output channels, for example, routing data to a serial port. Its implementation is based on Symbian d_exc.exe, which has been explained above. Basic functionalities of d_exc_mc.exe are kept same as originally implemented in d_exc.exe. Panic data and OS status are collected via the Symbian Minimal Kernel debug agent (MinKda). With moderate modification to MinKda to enable capturing software reset reason code, MinKda has been built into mobilecrashdriver.ldd. The file extension LDD is the abbreviation for Logical Device Driver in Symbian OS. The driver is loaded while executing the code of d_exc_mc.exe to collect the required panic data.

MinKda is able to capture the panicked process and the panicked thread. It can save the run-time loaded DLL lists and dump the call stack of the panicked thread. It can also store the processor registers at the time when a panic happens. By analyzing the collected data, developers can possibly locate and fix the bug. Besides the information of panic, d_exc_mc.exe collects the product specific information, which includes product type, device IMEI code, and the firmware version. This data together with the data collected by MinKda are then written into a binary file with pre-defined format. The generated binary file is commonly referred to as crash binary file.

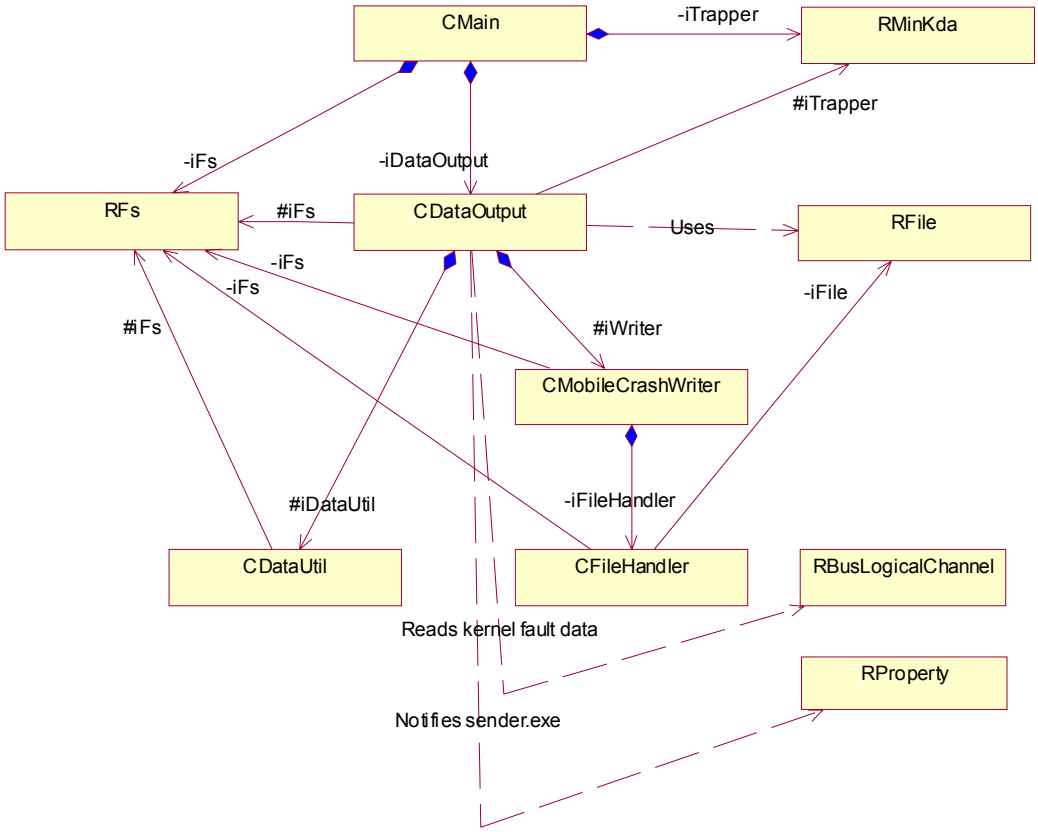


Figure 11 UML class diagram of d_exc_mc

Figure 11 presents the UML class diagram of d_exc_mc. The application framework is encapsulated in the CMain class, which has member variables as handles to debug agent and file session. The Symbian R-prefix classes indicate resources, and are usually defined by the OS as handles to resources. In the class diagram the RMinKda represents the debug agent and the RFs represents the file session. CMain initiates a trap to capture the Symbian user side panics. Once a panic is captured, it passes the data collecting procedure to its member variable iDataOutput that is a reference to the object of CDataOutput class. CDataOutput class has two member variables as references to object of CDataUtil class, which collects product specific data, and object of CMobileCrashWritter class, which writes the collected panic information and product information to a file. The file handling class CFileHandler defines data and file manipulation functions. Several classes in the class diagram are Symbian C-prefix classes, which inherit CBase class defined in e32base.h. All the C-prefix classes are customized classes for Mobile Crash project, and the R-prefix classes are defined by the OS. Readers without Symbian programming knowledge may feel difficult to follow the naming convention used, but it does not affect understanding the class relationship presented in the UML diagram. Further information in Symbian development can be found in Stichbury's book [Stichbury, 2005].

In the UML class diagram, class RFs defines a handle to File Session and functions to operate File Session. It has been referenced nearly by all Mobile Crash customized classes. This is for the reason file session consumes much system resource, and therefore sharing one file session within one application improves overall performance. The only file session used in d_exc_mc.exe is opened in the object of CMain class and passed as a reference to other objects. In addition, RFile defines a handle to every single file that is opened within the file session. In a usual case, many files are opened, operated and closed within one file session. When programming, this means one instance of RFs is created to open the file session, then as many as instances of RFile are created on demand to operate every single file used in the file session.

The CDataOutput class has also defined functions to read Symbian kernel fault information stored in flash memory or RAM memory. Kernel faults are trapped and relevant information for debugging are collected and written to protected memory partition by the Crash Debugger / Crash Logger framework, which resides on the Symbian kernel side. The framework will be introduced in Section 6.6. CDataOutput class uses RBusLogicalChannel derived class to open the channel between the Symbian user side and Symbian kernel side to access the stored data. After kernel fault happens, debugging data are collected by the Crash Debugger / Crash Logger framework, and then device reboots itself. The d_exc_mc.exe is started during device booting sequence, and stored kernel fault information are retrieved and written to a crash binary file.

Once a crash binary file has been generated, next is to transfer the file for parsing and decoding so to get the debugging data out of the file. The transmission can be done via short messaging service. The application sender.exe is developed for this purpose, and will be introduced in Section 6.7. The d_exc_mc.exe needs to notify the sender.exe there is a crash binary file ready to be sent. The communication between these two threads is achieved via the Publish and Subscribe framework, which is defined in Symbian OS v9. This framework allows setting, monitoring and retrieving system-wide variables (in programming terms, defined as Properties) to provide a mechanism for inter-thread communication (ITC). The d_exc_mc.exe sets the variable when there is a crash binary file ready. The sender.exe monitors the same variable, and starts the transmission procedure once detecting the variable has been updated. In programming terms, meaning that the d_exc_mc thread publishes a property and the sender thread subscribes the same property. In the class diagram RProprty class provides the functionality of Publish and Subscribe framework.

### 6.6 Log Symbian OS kernel fault

Symbian kernel faults are handled by the crash debugger / crash logger framework. The crash debugger is an interactive utility that collects the fault information and presents the data depending on the executed command. The crash logger is a non-interactive

utility that logs the fault information to the pre-reserved location on permanent storage. Both of these utilities rely on common monitor functions to dump various fault information such as processor register sets, thread stacks and kernel object containers. These features have been encapsulated in one kernel extension module exmoncommon.dll to avoid duplicating code size. Symbian kernel extensions are special DLLs that are loaded when OS boots. A kernel extension is entirely kernel-side code, without necessarily providing any user side API. This is different as the Symbian device driver, which is also built as special DLLs but always includes a user side API. Examples of kernel extension are keypad and touch screen implementations. They are loaded at OS boots and only interact with the kernel.

Either crash debugger or crash logger is an implementation of Symbian kernel fault monitor. They both inherit the Monitor class, and are built into individual kernel extensions. In Symbian OS release, kernel extension exmondebug.dll encapsulates crash debugger implementation and kernel extension exmonlog.dll encapsulates crash logger implementation. Either or both the monitors could register with the common module exmoncommon.dll, and be called upon a Symbian kernel fault is trapped. The exmoncommon.dll must be placed in ROM image before either or both exmondebug.dll and exmonlog.dll. Upon a kernel fault, the control is first handled to the monitor that is first registered with the common module, then is handled to other monitors depends on their sequence of registration with the common module.

In Mobile Crash, a customized kernel fault monitor has been implemented. It reserves a partition of SDRAM memory for crash data storage. Upon kernel fault, it collects required kernel fault information then logs data to the reserved storage according to the pre-defined crash binary file format. On the next OS boots, the d_exc_mc.exe performs the reader functionality to retrieve the data, and then writes data to crash binary file. The customized monitor rewrites part of Symbian crash debugger implementation, and in addition provides two new classes. Therefore it's able to reuse the crash debugger code to register with the common module exmoncommon.dll. The customized monitor is built into binary target exmondebug.dll and replaces the Symbian crash debugger. The control is handled to exmondebug.dll upon kernel fault, but instead of executing the original interactive crash debugger the SDRAM featured logger operation is executed. The customized monitor has its own driver implementation to perform logging data to SDRAM memory, as when kernel panics no particular part of the Symbian OS could be assumed to work properly. The driver also provides the use side API, which is called by d_exc_mc.exe to retrieve data from the reserved memory.

Two binary modules are implemented to log kernel fault information. One is the customized monitor exmondebug.dll, which includes the modified CrashDebugger class (from Symbian OS release) as well as two new classes DCrashData and DSdramBuffer. The other is the combined kernel extension and device driver binary module

sdramcrashloggerext.ldd, which provides dual functionalities. As a kernel extension, it is loaded during the device boots and it initiates the crash debugger module by providing SDRAM physical address range. As a device driver, it provides read / write functions on the reserved SDRAM partition and the corresponding user side API to call these defined functions.

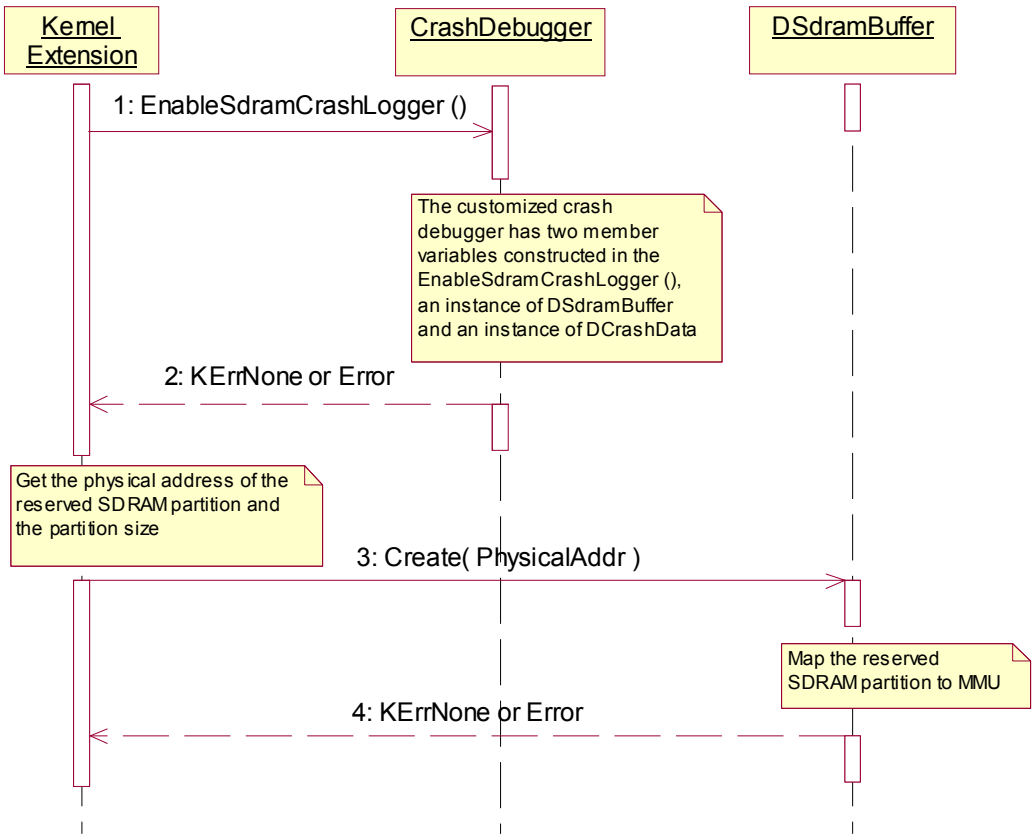


Figure 12 Sequence of initiating the customized monitor

In device booting sequence, the binary modules exmoncommon.dll and exmondebug.dll are loaded first. Then the kernel extension implemented in sdramcrashloggerext.ldd is loaded. It provides the physical memory address range and enables the customized crash debugger, after which the crash debugger is armed to handle kernel fault. On the init sequence diagram (Figure 12), crash debugger creates an instance of DCrashData and an instance of DSdramBuffer. The DCrashData provides a routine to collect panic data and write to a temporary buffer according to crash binary file format. The DSdramBuffer provides SDRAM read / write functions and CRC verification on SDRAM stored data.

Upon a kernel fault, Symbian OS calls *Monitor::Init ()* to first pass the control to the common monitor module, which further passes the control to the customized crash debugger. Kernel fault information is collected by the member variable of type

DCrashData, and then written to the reserved SDRAM partition by another member variable of type DSdramBuffer. After the kernel fault has been handled, crash debugger returns a variable containing restart type to the common monitor, which then forces the Symbian OS to reset (device to reboot). The logged data is retrieved by the d_exc_mc.exe process after device reboots. (See Figure 13)

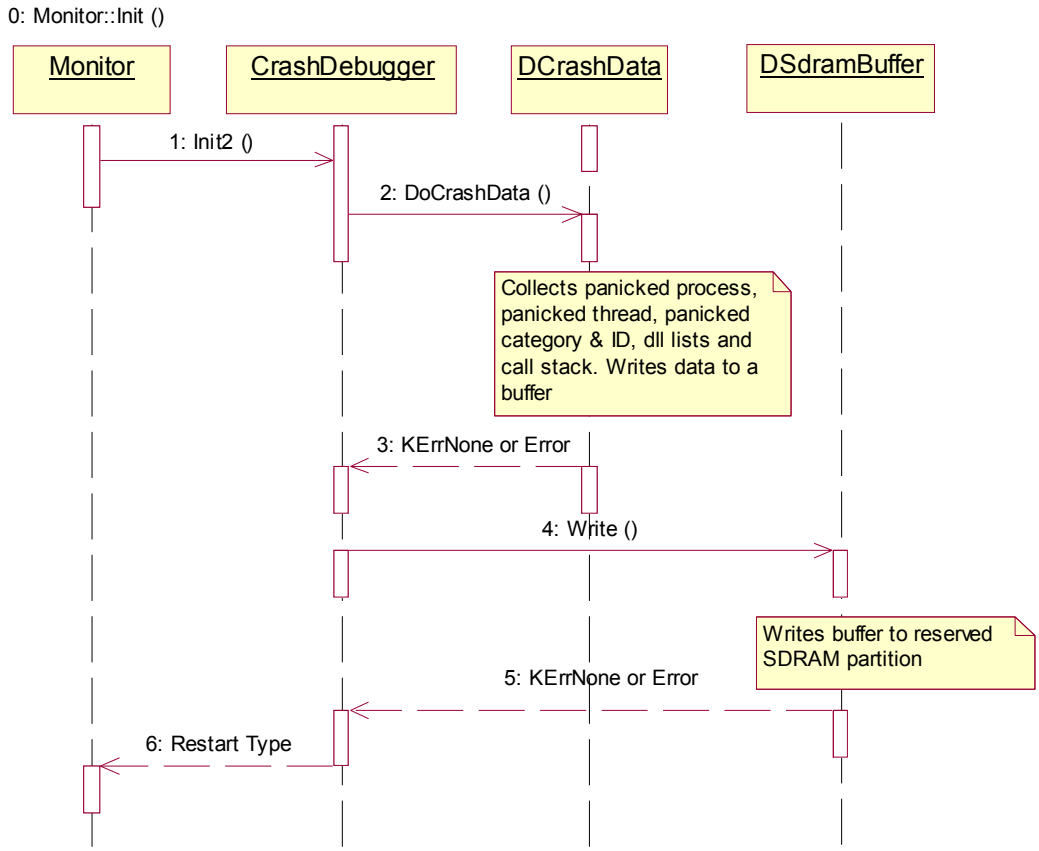


Figure 13 Sequence of handling the kernel fault

Data retention in the reserved partition of SDRAM memory is one concern. After the kernel fault has been processed the monitor requests the device reboots, therefore the logged data should not be lost while the device is rebooting. The data retention requires the underneath hardware to support a self-refresh SDRAM subsystem. The other concern is the reserved partition should not be used by any other components but only exmondebug.dll and d_exc_mc.exe. This requires configuring the device bootstrap so that the reserved partition is marked not being allocated for ordinary OS use.

### 6.7 Transmit crash binary file

Crash binary files are stored to device file system, either on memory card or device flash memory. Later they are transferred to PC to be decoded. For developers, transmission takes ways of USB connection from device to PC, or direct file copy from

memory card to PC. Most developers concentrate on a few software components on limited device platforms at a time, and the amount of crash binary file handled is fairly small. Hence transmitting crash binary files via local connection is flexible and reliable. The complete file can be transferred without cutting any information because data amount is not any concern for local connections. Usually one crash file can hold up to 20KB of data. On PC crash files are parsed, and dumped call stack are decoded to uncover the crash time collected data, such as processor register sets, DLLs list, and on stack pushed functions pointers (refers to Section 6.8). This information helps developers to pinpoint any possible software defects.

One outstanding benefit of Mobile Crash is to trap Symbian OS panics while device is being used ordinarily in stead of in R&D environment. This means the product has reached certain maturity and testing can be conducted by forming a user group, where members are selected from volunteers willing to use the test device as their personal mobile phones. In such test some software components may crash after certain combination of user interaction events. Therefore panic is trapped by Mobile Crash, following the collection of panic data and processor states, and crash binary file is written in the end. It's not expected from end users to transfer crash binary files to responsible engineers, besides in many cases it's not clear what underneath software components have caused the crash.

In the above case, crash binary file transmission should require minimal user action, and preferred to be executed immediately after a Symbian OS panic. This aims to reduce the error handling time, so to fasten the procedure of product-to-market. The other reason behind minimizing the transmission delay is to provide a realistic statistic of product maturity, i.e., to collect as much and fast as possible device failures on field and analyze these defects within an evaluation period. If certain software component ranks high frequency of crashes, time and effort can be prioritized to improve the quality of that component.

Transmitting crash binary files automatically via SMS does not require much user action, and SMS itself has been a mature technology for a while that would rarely break. The auto SMS sender application developed for Mobile Crash gets noticed when a panic is trapped and a crash binary file is written. After that it starts sending crash binary file in short messages. Data load and consequent network traffic costs are the concerns of SMS transmission. In practice, one SMS message can hold up to 140 octet byte (that is 8 bits for one byte, to distinguish with the ordinary text message using 7 bits for one byte). Considering the average crash binary file size is over 10KB, more than 70 SMS messages are sent if not reducing file size.

Careful studies have been conducted on crash binary files, aiming to remove the part of data that do little help for developers to debug the trapped Symbian OS panics. It is found that the dumped call stack and run time loaded DLLs take much space

compared to rest of the crash binary file. Regarding to call stack, three findings are outlined here. First, the stack section above the stack pointer unveils the function calling sequence before Symbian OS panics, and this part is usually of the concern to debug. Second, Section 6.3 has pointed out that automatic variables, temporally allocated data segments, and function pointers from previous successfully completed function may also stay in stack. Third, software is different and there is no ultimate solution to work out which stack word is the valid function pointer before the crash. However by mapping stack words to Core OS memory address space and run time loaded binary modules memory address spaces, stack word that lays within one of those address ranges can be assumed as a valid function pointer.

The auto SMS sender application executes the operation of mapping stack words to concerned address ranges. Only the assumed function pointers are remained on stack, the rest are deleted. The original dumped call stack often contains hundreds of stack words, and apply such an operation to the complete stack slows down the overall performance. Because only the part above the stack pointer is of interest to analysis and debugging the trapped panic, the sender application cuts a section above the stack pointer out of the complete stack for the mapping operation. The section length can be configured, in the current implementation it is of 100 stack words length.

Apply the mapping operation for a stack section not only reduces the stack length sent over the air, but also reduces the DLLs list length. The collected DLLs list contains the run time loaded binary modules including DLLs and executables. Each module name together with its path and memory address range are stored to crash binary file in text ASCII format, and occupies up to 30 bytes space. Often the list contains over 10 binary modules, which would have the data load consuming 2 to 3 SMS messages. If there is no stack word mapping to some modules in the list, those binary modules are considered irrelevant with the trapped panic and removed from the DLLs list.

The crash binary file size can decrease to less than 1 KB after cutting down data from call stack and DLLs list. Despite the significant size reduction, one file is sent with multiple SMS messages. Considering a real life circumstance, files from different devices are sent to a central receiver for decoding and analysing. An identification mechanism needs to be introduced to distinguish messages from different files. Besides, the GSM network might cause delivery delay, or even message lost. This requires the receiver having the time-out check for those pending crash files due some messages from the file do not arrive. Furthermore, even all messages originated from the same file are received; they may not arrive in the same order as they are sent. Each message has to bear the information of its order in originated crash binary file, when dispatching from the device.

The sender application defines a customized identification protocol, and appends a header for each message. The header contains data to uniquely identify the originated

crash binary file, the order of the very message detached from the file, and the total number of messages sent for the file. Messages generated from the same file bear the same identifier, and are concatenated to the file according to their indexes (message detaching orders from file) at the receiver side. The receiver implements the same protocol to recognize messages from different files, and then temporarily store messages with same identifier to a linked list data structure. Upon the moment all messages with same identifier arrive, they are concatenated to a crash binary file. If at least one message do not arrive within the predefined time, all arrived messages bearing the same identifier are deleted.

The header also contains the length of message data section. This is for the receiver to read the exact data length so to prevent invalidating the crash binary file from reading extra data. The defined header is 8 bytes long, which leaves 132 bytes available for the data section. Often the last message originated from file has various lengths, but other messages have the data section fulfilled. The crash binary file has the CRC written at the last 4 bytes. When a file is reassembled, the receiver performs operation to calculate the CRC and compares the value with the one recorded at the end of the file. If the two CRC values do not match, then the file is considered to be corrupted during the transmission and gets deleted.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 ... 139 |
|---|---|---|---|---|---|---|---|---|
| FG | Identifier | | | | TOT | IDX | LEN | Message Payload ... (Up to 132 bytes) |

Figure 14 The message header defined by customized protocol

Figure 14 presents the message header used for the data transmission. The maximum length of the 8-bits byte encoded message is 140 bytes. The first byte tells whether the message is a part of concatenated message or a stand alone one. The value of FG can be 0, meaning a stand alone message. In this case, the message does not use the following 7 bytes of the header, and consequently leaves 139 bytes long data section. Often the message is a part of a concatenated message, and FG has the value of 1. Following 4 bytes in the header is the identifier to distinguish which crash binary file the message originates. The next three bytes tell the total number of messages generated from the file (TOT), the index of this very message detached from the file then sent from the device (IDX), and the length of the data section (LEN), respectively. The rest of the message contains data of crash binary file, and it is marked as the message payload on Figure 14.

The sender application is built to a binary module sender.exe on Symbian OS. The sender.exe is started with other Mobile Crash binaries at device boots. An ini file is defined to include the receiver number, and it is stored on the device file system. By placing a different ini file on the memory card then rebooting the phone, the receiver

number is reset on the sender. With the current implementation, 2 minutes delay is placed on the sender.exe main thread right after the thread is started. This is due to Symbian OS notifier server starts very late during the device boots. And sender makes use of the notifier server to pop up a text box informing end user every time a panic is trapped and messages are sent.

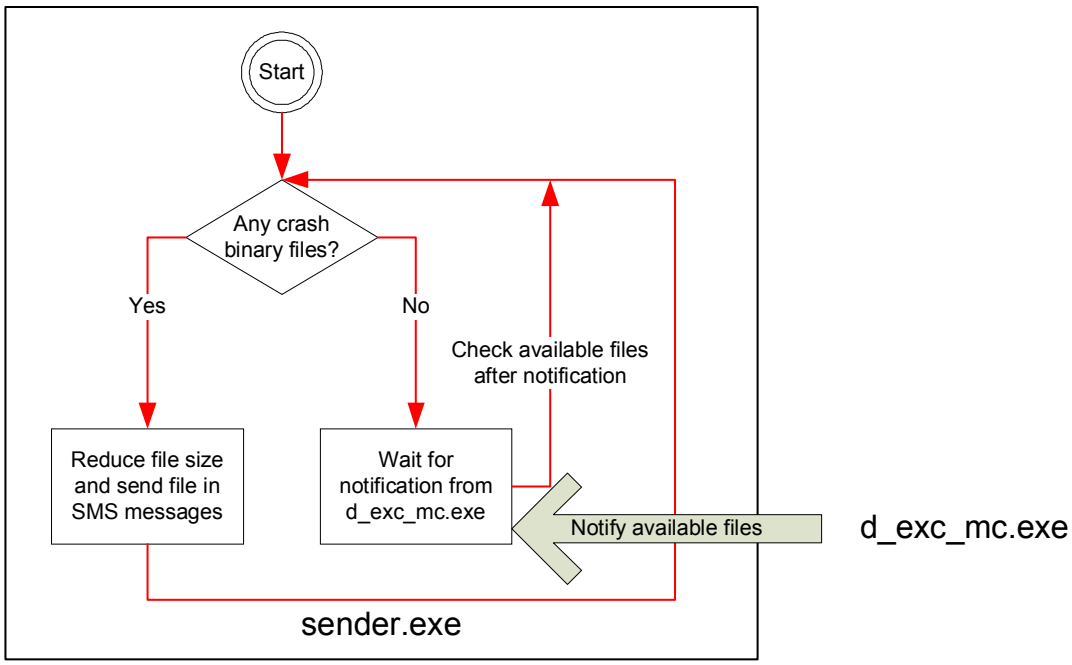


Figure 15 Inter thread communication between d_exc_mc.exe and sender.exe

Figure 15 presents the flow chart of the sender.exe execution. It first scans the configured directories for crash binary files that are generated by the customized crash debugger, which resets the device after kernel fault is trapped. In the current implementation, the target directories are memory card root and the C:\data\. Any crash binary files left in those folders are processed after sender.exe starts. Then it runs in a stand-by mode, and waits notification from d_exc_mc.exe. Section 6.5 explains that the d_exc_mc.exe and sender.exe make use of Symbian Publish and Subscribe framework to communicate. The d_exc_mc publishes property once a panic is trapped and a crash binary file is written, meanwhile the sender subscribes and monitors the property to take actions of processing crash binary files when detecting a property update event.

In practice, a central receiver is set up to collect crash binary files, decode crash data, and further generate a report for each file. Product maturity is also analyzed on such a receiver. It is not only used for one or two products, but rather for several product platforms covering dozens of products. More about decoding and analyzing are discussed in Section 6.8. SMS message transmission is one way of collecting crash binary file for the central receiver. Besides, files are also transferred via FTP over the LAN. This requires crash files are first routed from smartphone to the nearest workstation that is connected to the LAN, and that workstation should have the FTP

client software configured to connect to central receiver. Such workstations are scattered around to facilitate transmitting crash binary files.

For the purpose of easy setting up on workstation, a software application – Hoover.exe that packages FTP client configuration is developed. It automates the transmission by simply connecting the smartphone to workstation via USB cable. A procedure window is then presented to notify end users when the transmission is done. Crash binary files collected by this way do not need to cut down any data, the complete call stack and DLLs list can be transferred along with the file. On the other hand, it depends on end users whether to transmit the file and when to transmit, which may cause some crash data are not collected or delayed to decoding and analysis.

Tracing application execution has been often used while debugging embedded software. Nokia has developed its proprietary protocol that collects the print type trace via serial port. If the smartphone is connected to the tracing hardware, the content of crash binary file can be routed through the trace interface to serial port, and then collected by the trace representing application. This R&D feature is provided by d_exc_mc.exe for data collected due Symbian user panics. Same feature is provided by the customized crash debugger for data collected due Symbian kernel faults. Because traces are collected on real-time, crash data transmitted in this way has the minimal delay. It facilitates developers debugging and analyzing software failures without adding any new hardware or software on the existing R&D environment.

### 6.8 Decode crash binary file and operate database

On central receiver, crash binary files are parsed and decoded by a software application – Selge.exe. Parsing file is executed according to the pre-defined file format. A crash binary file is composed by a header and various numbers of items. The header contains the file format version, the timestamp when file is written, and the number of items followed. The file format version field is reserved to distinguish various formats applied, and to be checked while paring files. However, in the current implementation only one available file format is being used, and the version data field has no effect on how crash binary files are parsed. Each item includes three fields, which are the item ID, the item type, and the meaningful data. The ID is a 16 bits unsigned integer and tells what is the data recorded in item, for instance 0x0005 indicates the item stores the Symbian OS panic category, and 0x0009 indicates Program Counter. The type is also a 16 bits unsigned integer and tells what data type is applied for the stored data. Often ASCII type or integer type are used in crash binary file. The last field is the stored data.

Most of the collected crash data are written to crash binary files in various composite formats. Text ASCII type is one of the simplest composite formats. It has an additional header to determine its length. When parsing an ASCII type data item, the length is first retrieved then the desired length of data is read from a crash binary file.

More complicated type like DLLs List, which begins with a header to determine the number of DLL Items followed. Then each DLL Item contains Start Address field, End Address field, and Name field. Among those, the two address fields are written in unsigned 32 bits integers, and the Name field is written in Text ASCII type. Both the file writing procedure and file parsing procedure are developed with very much concern of the file format specification, so to retrieve collected crash data from crash binary files after transmission.

The cyclic redundancy check (CRC) is computed against the file content to produce a checksum – a 32 bits unsigned integer. The checksum is then attached at the end of crash binary file. The checksum is verified by the recipient during the file transmission, and by the selge.exe application before the file is parsed. If detected the checksum has been changed, which indicating the content of crash binary file is invalided, the file would get deleted.

The file format specification defines a special type of crash binary file, named as registration file, which contains only the product related information but not any valid crash data. A registration file is generated by the d_exc_mc.exe only at the first boots after a software update has been flashed (loaded) to the product. The registration file is treated as an ordinary crash binary file and is transmitted to the central receiver. The idea of writing such a file is to know how many products have been updated with the new software release. Then the number is used to facilitate statistic calculation to evaluate software release maturity.

Section 6.3 has covered decoding Symbian OS stack in details. The central receiver has applied the same technique to decode the dumped call stack according to the ROM image symbol file and ROFS image map files. The decoding procedure refers to a symbol table and various map files generated when the flashable firmware image is built. Considering the central receiver has to work on crash binary files originated from different product software releases, the decoding procedure is configured by loading different symbol tables and map files according to ROM image identifier (refers to ROM ID in table 1) parsed from the crash binary file.

Next we present some background knowledge on how the flashable firmware is built. The complete firmware is encapsulated into one single image file. The term flashable is referred as the image file is first loaded to product flash memory, then Symbian OS is booted from the flash memory. A flashable image combines two images in a single file. One is the ROM image containing the Symbian Core OS, which is completely loaded from flash memory to a dedicated section of RAM memory during device boots, and permanently stays there. This section of RAM memory is read-only accessed by the ROM file system. The other one is the ROFS image containing mostly application or utility binary modules, which are loaded on demand to RAM and

removed when they are not needed. The ROM ID is a 32-bits checksum word computed against the ROM image data block. It uniquely identifies the flashable firmware.

When the ROM image is built, a file (named symbol file) contains the symbol table is generated meanwhile. The symbol file serves to map the memory address scopes to binary modules included in ROM image. Once the ROM image is updated, the symbol table is updated as well. The counterpart map files are generated while the ROFS image is built. Each application or utility binary module included in ROFS image has its own map file. Section 6.3 presented decoding stack word by calculating offset to the start address of run time loaded binary module and mapping the offset to the module map file. See also Figure 8 in Section 6.3.

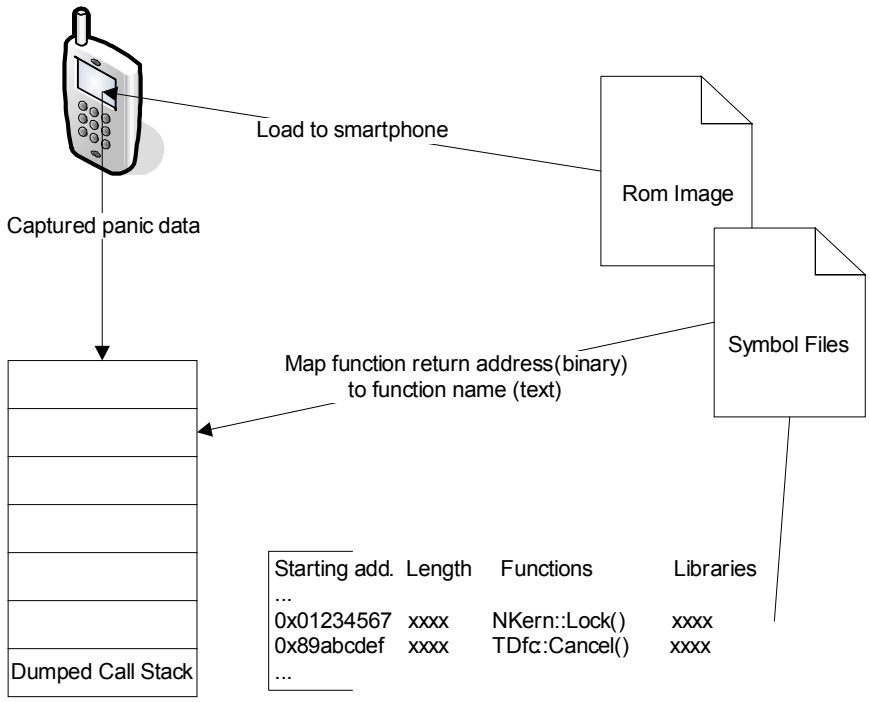


Figure 16 Decoding stack word from ROM image symbol file

Figure 16 presents the stack decoding procedure from the ROM image symbol file. The call stack has function pointers pushed onto the stack. If a stack word happens to lay between the memory address ranges of certain function, it is then mapped to that function. Mapping stack word to ROM image symbol table is simpler than mapping to ROFS map files. It does not require comparing the stack word to the run time loaded DLLs list then calculating the address offset, instead the stack word can be mapped to the symbol table straight forward. This is because the ROM image permanently resides on memory, and is always loaded to the dedicated memory section. Decoding the dumped call stack enables developers to track the function calling sequence of the executing thread before panics.

After crash binary files get decoded, data are stored to various categories in database. The decoded call stack can be queried by a reference number, which is built up by the crash timestamp. To facilitate querying database for crash binary files and making statistic analysis, a Mobile Crash web UI is developed. The web UI automates the querying procedure in certain time interval to fetch the latest decoded crash binary files from database and present those on a web page. The web UI provides various operations to sort crash binary files and query crash binary files qualifying certain key values. For instance, end users can query crash binary files, which are originated from a specified firmware release on a named product.

The web UI also provides statistic analysis based on the named product or the specified firmware release on a named product. Analyzing crash count on software components would point out which modules have been crashed more often than others on certain firmware release. Time and effort can be prioritized on fixing those modules to effectively reduce the total crash count on the firmware. Figure 17 presents the analysis of crash count on software components for certain product. The analysis is made on a continuous 6 firmware releases of the selected product. The product firmware is released in fixed time interval, for example biweekly, to add new components and place fixes for the errors found from the previous releases. On the diagram, each column represents one firmware releases, and each row represents one software component. The number on the table indicates the crash count of the selected component on the selected release. The table tells, for example, the component Imsrvapp has only crashed on the latest release but not any previous releases. The reason behind could be either the module was recently implemented and has not been well tested, or a recent update on the module brought up those crashes. In addition, if libraries or application frameworks, on which the component Imsrvapp depends, have been changed recently, it may consequently result in the component crashes.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| RealPlayer | 0 | 3 | 5 | 1 | 0 | 14 | 23 |
| Imsrvapp | 0 | 0 | 0 | 0 | 0 | 11 | 11 |
| Maps | 0 | 11 | 1 | 3 | 4 | 6 | 25 |
| HTIFramework.exe | 0 | 0 | 6 | 0 | 0 | 4 | 10 |
| Video tel | 0 | 0 | 3 | 7 | 0 | 3 | 13 |
| Download music | 0 | 0 | 0 | 0 | 8 | 3 | 11 |
| BrowserNG | 5 | 27 | 14 | 17 | 9 | 3 | 75 |
| Multimedia message | 0 | 8 | 1 | 3 | 0 | 2 | 14 |
| Media Gallery | 0 | 13 | 0 | 16 | 0 | 2 | 31 |
| Idle | 0 | 0 | 1 | 6 | 2 | 1 | 10 |
| File Browser | 0 | 1 | 1 | 1 | 0 | 1 | 4 |

Figure 17 Analyzing crash count on software components

Mobile Crash is intended to be used by developers to track down the captured call stack, then possibly locate software defect on code lines. When it is applied to software

testing and integration phase, the person who encounters Symbian OS panics is instructed to transfer crash binary files to the central receiver for decoding and analysis either by SMS messages or FTP connection. It is also recommended for software testers and integrators to write a short description of what operations result in the panic. Then the developer who is about to handle the crash can reproduce it, if needed. The Mobile Crash web UI provides a way for writing such a note. After the concerned crash binary file has been decoded and stored to the database, it can be queried by any form of the combination of IMEI, originated device phone number, and file reference number from the web UI. It is usually easy to identify the exact crash binary file by checking panic timestamp if multiple results are returned from the query. Individual crash binary file can be opened as a pop-up window from the web UI. Software tester or integrator can then edit the user comment field on the window to describe the pre-condition of the encountered crash or provide any information that may be helpful for the developer to solve the problem.

The alternative way to record the crash pre-condition is to place a test set ID field in the crash binary file. Each ID maps to one or more test cases. Developers can then refer to the test set ID to reproduce the crash with defined test cases. Before software testers make any test on device, a specified file storing the test set ID should be modified to match the test to be executed on device. The file is then saved to the designated folder on device file system. If panic happens during the test, d_exc_mc.exe reads the file and records the test set ID to crash binary file. The test set ID is also presented with other data on the crash binary file pop-up window when browsing from the Mobile Crash web UI.

### 6.9 Ideas to improve Mobile Crash

Even though the practice of test set ID and user comment have been applied to Mobile Crash to best record the panic pre-condition, there lacks a practice to record OS level system events before panics. Such events like reserving / releasing resources or starting / stopping threads could provide valuable information for software defect analysis. On the other hand, it very much depends on software testers and integrators to report panic pre-condition via the current implemented practices. If user comment is used, the described pre-condition can sometimes be unclear or incomplete for developers to reproduce the panic.

A better solution would be to automate the panic pre-condition record procedure, and extend the recorded events to contain both user interaction events and OS level system events. All user interactions like open / close applications and the corresponding OS level activities like starting application thread, scheduling threads and reserving system resources can be recorded to a dedicated memory buffer. If panic happens, certain numbers of latest recorded items are written to the crash binary file. Regarding

to the automatic events record procedure, issues of buffer size and event logging need to be carefully studied. A FIFO queue type of data structure can be used to record events. If the size of the queue is defined to contain any 10 items, then the latest 10 events are recorded. An event registered listener can be used to get a notification when the specified event happens, and further action can be taken to log the event to memory buffer. If Symbian OS does not provide such event listeners, then customized implementation need to be applied.

The automatic panic pre-condition events logging is the first proposal discussed to extend the Mobile Crash features. If this practice will be implemented on Mobile Crash, nearly all existing software components would need to be updated. Briefly outlining the changes on the affected components are described here. The d_exc_mc and the crash debugger modules should implement logging events to memory buffer, and writing them to crash binary files. The file format specification should define the format for the logged events in crash binary file. The decoding application and database structure should recognize those logged events and store them to a new database entry. In addition, the web UI should be modified to present the recorded events for each crash binary file.

While more data are added to crash binary files, transmitting via SMS messages will demand more messages to be sent and received. These appear to be error-prone, as if one message is lost or invalided during the transmission, the complete crash binary file will have to be discarded. The second proposal is to improve the transmission media, for example by sending file encapsulated in a GPRS data package instead of SMS messages. This practice requires the GPRS related data communication works reliably on the product, and would rarely break when crash binary files are frequently sent over it. If this update applies to the Mobile Crash, the sender.exe module should then be modified to transmit crash binary file via GPRS data package. Besides, the SMS Gateway component should be replaced by a GPRS Gateway, which handles receiving and validating data packages, then routing crash binary files to decoding application.

Nowadays, Mobile Crash is being used as a R&D tool during the product development and testing phases. There lacks a similar tool that can be used on devices sold on market. The tool aims to store crash data on device whenever a panic happens. Those saved crash data can be analyzed after end users return the faulty products to retailers. This idea becomes the third proposal to extend Mobile Crash features. The way crash data are stored and retrieved should be changed for adapting to devices sold on market. Crash data should not be visible by file browser kinds of applications so that end users would not notice crash data on devices. This can be achieved by writing crash data to a reserved memory partition that is prohibited from being used by other OS operations and hidden to file system. Later the partition can be accessed by a dedicated memory reader application. The size of the partition should be limited to not have much

impact on the total memory budget. Only limited numbers of crash binary files can be written to the reserved partition. The crash data items should be carefully selected so that the most significant data are recorded to crash binary file. This is also for reducing the file storage memory consumption.

The last proposal discussed in the chapter is to standardize the way Mobile Crash components accessing Symbian OS application APIs. The customized crash debugger module has modified the Symbian OS source code. This generates a risk that crash debugger may not work properly if Symbian makes an update on the concerned code. A better way would have the basic crash monitor functionalities abstracted and encapsulated into a binary module, and maintain a constant application APIs. As long as the Symbian code updates do not change the API, the crash debugger will work as expected. Changes like proposed would need to cooperate with Symbian.

At the time of writing this thesis, Symbian has made modification on how data recognizer is loaded to the OS (recognizer appoints a default application to open files of the specified MIME type). Before the update, all MIME type data recognizers are loaded during the device boots. Many developers take advantage of this feature to start their own applications inside an empty recognizer during the device boots. Nearly all Mobile Crash binary modules are started also in this way. After the Symbian update, MIME type data recognizers are loaded on demand, in order to reduce the device boot time. This change has brought requirement to develop a new mechanism to start Mobile Crash binary modules during the device boot. This case has also explained the importance to standardize accessing Symbian APIs, to prohibit applications from working unreliably due Symbian code updates.

# 7. Conclusion

Till now the reader should have a general understanding of applying Mobile Crash to debug Symbian OS powered embedded system. This chapter shortly summarizes the key points covered on the thesis and reminds the reader the concerns when developing and debugging Symbian OS applications.

Embedded system development requires careful design and implementation. From one perspective, the hardware design needs to be compact, on both size and cost, to complete the desired functionalities. This is because embedded systems, for example smartphones, often target on volume market. Cutting production cost in a small portion results in significant profits. From the other perspective, the firmware development on embedded system needs to be reliable and more error proof than software designed on workstation computer, because embedded systems often operate continuously in weeks or months without frequently reset the system. Beside to it, some embedded systems are placed out of the reach of ordinary maintenance, or cannot be updated as easy as downloading and installing a service pack compared to software on workstation computers.

Symbian OS is designed to target resource constrained devices like embedded systems. The native developing language for Symbian OS – C++ has also been tailored to suit the limited available system resource of devices. Applications developed on Symbian often have dependency on various services provided by OS, which reflects the fact that much of the OS designs are based on Client-Server architecture. Symbian remains as the most popular OS on smartphome market at the time this thesis is written. But Symbian development often requires sharp learning curve for developers as it invents specialized ways to write C++ code. In addition, support on developing tools and availability of example code are not as good as Windows or Linux development.

Developing Symbian application on emulators has the advantage of integrating to IDEs and utilizing the rich debugging features of IDEs. Emulators are often freely available from device manufacturer technical support site, and remain as the first choice for small third party software development firms and individual developers. Using emulators may shorten the learning curve for experienced developers to move from standard C++ development to Symbian C++ development. On the other hand, emulators are not suitable for device drivers or OS services development, for the reason emulators do not have identical software modules on the host platform as those adapted to embedded system hardware. Symbian software developers need to bear in mind that emulator cannot replace target device. Emulators do not guarantee to perform identical behaviours as applications run on target devices. Symbian software should always be tested on target device before they can be released to market.

Tracing application execution on target embedded system provides run time information, including key variables and application branch coverage, for developers to analyze. It applies both to system level software development like device drivers or application development above the UI framework. Tracing embedded system often requires special hardware and software. Device manufacturers have their proprietary tracing protocols designed, which cannot be accessed by third party software developers. Tracing are mostly used on Symbian Core OS development. Time and effort spent on building and reloading firmware to target device while repeatedly tracing application to debug may overshadow the developing efficiency. Hence, the solution to dynamically enable / disable certain groups of trace on target device reduces the unproductive time of applying tracing on embedded system development.

Mobile Crash as a post-mortem debugging utility, which automates reporting the Symbian panics, contributes detecting software failures during the software integration and testing phases. It captures Symbian OS panics and exceptions, collects data of panic and processor states. The collected data provides valuable information to pinpoint suspicious code in the panicked software component. These data together with a brief explanation of what operations have been carried out before the failure help developers to reproduce the error and debug the suspicious code. From problem-solving point of view, the benefit of applying Mobile Crash is effectively limit the size of code developers would need to investigate to pinpoint the bug. Yet Mobile Crash is not a run time debugger, and cannot be expected to always precisely point out the problematic code. Developers should combine the use of Mobile Crash with other debugging methods to solve software failures on Symbian OS.

As software failures are captured, reported, and decoded automatically, Mobile Crash provides a convenient way to process and present failure information. Software integrators and testers do not need to gain any special knowledge to use the tool. The collected crash binary files are decoded on a central server and Mobile Crash web UI serves as a portal to easily query the decoded data. Often software developers would not need to do the decoding manually unless panics or exceptions are generated on purpose and crash binary files are analyzed locally on a R&D environment.

In addition, crash statistics can be calculated from the collected crash binary files on the central server. The result can be used to, for example, pointing out the unreliable components or evaluating the maturity of certain firmware releases. In the end using a few words to summarize Mobile Crash, it offers a turnkey solution to automate capturing, reporting, decoding, archiving and presenting Symbian OS software failures, in order to improve smartphone firmware development productivity.

## Reference

[Allen, 2005] Lee Allen, Advanced Mobile Operating Systems: Comparative Analyses & Forecast. The Diffusion Group, November 2005. Available as http://www.tdgresearch.com/product.asp?itemid=73&catid=33

[ARM, 2006] ARM, ARM processor and development documentation, Available as http://www.arm.com/documentation/

[Canalys, 2005] Canalys Analyst Firm, Worldwide smart phone market soars in Q3, October 25, 2005. Available as http://www.canalys.com/pr/2005/r2005102.htm

[Murphy, 2004] Brendan Murphy, Microsoft Research, Automate Software Failure Reporting, November 2004. Available as http://swig.stanford.edu/~fox/cs444a/readings/murphy_failureanalysis.pdf

[Ollila, 2006] Jorma Ollila, Speech given in the CTIA WIRELESS 2006 conference, April 6, 2006.

[OMAP, 2005] Texas Instruments, Wireless Solutions – OMAP Platform, Available as http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=11990&path=templatedata/cm/product/data/omap_2420

[Sales, 2005] Jane Sales, *Symbian OS Internals, Real-Time Kernel Programming*. John Wiley & Sons Ltd, 2005.

[Stichbury, 2005] Jo Stichbury, *Symbian OS Explained*. John Wiley & Sons Ltd, 2005.

[Symbian, 2006] Symbian Ltd, Symbian announces new pricing models to accelerate mass-market adoption of Symbian OS, February 8, 2006. Available as http://www.symbian.com/news/pr/2006/pr20063401.html

[Symbian OS Essential, 2005] SysopenDigia, *Symbian OS Essential* Training, 2005.

[Symbian v9.2, 2006] Symbian Ltd, Symbian OS Library for Device Creators, Online resources, reference guides and documentation for Symbian OS v9.2, 2006.

[TI, 2005] Texas Instruments, Software Development Platform for the OMAP2420 Processor, 2005. Available as http://focus.ti.com/pdfs/wtbu/TI_sdp_omap2420.pdf

[Wikipedia - ARM, 2006] ARM architecture in Wikipedia, Available as http://en.wikipedia.org/wiki/ARM_architecture

[Wikipedia - Embedded System, 2006] Embedded system in Wikipedia, Available as http://en.wikipedia.org/wiki/Embedded_System

[Wikipedia - Smartphone, 2006] Smartphone in Wikipedia, Available as http://en.wikipedia.org/wiki/Smartphone

[Wikipedia - Symbian OS, 2006] Symbian OS in Wikipedia, Available as http://en.wikipedia.org/wiki/Symbian_OS

# Appendix 1 - Glossary

| | |
|---|---|
| API | Application Programming Interface |
| ASIC | Application Specific Integrated Circuit |
| ASSP | Application Specific Standard Product |
| CPSR | Current Program Status Register |
| CRC | Cyclic Redundancy Check |
| EKA2 | Epoc Kernel Architecture 2 |
| FIFO | First In, First Out |
| GPRS | General Packet Radio Service |
| GSM | Global System for Mobile Communications |
| IDE | Integrated Development Environment |
| IC | Integrated Circuit |
| ITC | Inter Thread Communication |
| MinKda | Minimal Kernel debug agent |
| OS | Operating System |
| PC | Personal Computer |
| R&D | Research & Development |
| RISC | Reduced Instruction Set Computer |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| ROFS | Read Only File System |
| RTOS | Real Time Operating System |
| SoC | System on Chip |
| SMS | Short Message Service |
| UI | User Interface |
| USB | Universal Serial Bus |
| XIP | eXecute In Place |

## Appendix 2 - An Example Crash Binary File

This appendix is about to demonstrate how crash binary files may assist software developers to debug software defects. Product specific information presented here should not be disclosed outside Nokia. Therefore certain product specific data is replaced with *"Any Value"* phrase. Table 5 presents a crash binary file collected from Nokia N95 smartphone.

Crash Report

| | |
|---|---|
| Timestamp | 26.11.2006 10:13 |
| Crashed module | Gallery |
| Panicked Process | Gallery |
| Panic Id | 3 |
| Panic Category | USER-EXEC |
| ROM Id | *Any Value* |
| Software Name | *Any Value* |
| Software Info | *Any Value* |
| Variant Id | *Any Value* |
| Language | English |
| Program Counter | 8010ec53 <8010ec4d> 0008 User::WaitForAnyRequest() euser.in(.text) |
| Stack pointer | 0040ee90 |
| Stack base | 0040d000 |
| Available memory | 18124800 |
| Product type | *Any Value* |
| IMEI | *Any Value* |
| User Comment | Gallery stuck and crashed when viewed a broken TIF |

---

Call Stack

| | |
|---|---|
| Adis.dll | CAdisImageProcessor::EventLoopL() |
| core.symbol <8094695f> 0076 | CFbsBitmap::SetScanLine(TDes8&, int) const fbscli.in(.text) |
| Adis.dll | CAdisImageProcessor::ThreadMain(void*) |
| MGXUiBase.dll | CMGXAvkonViewImpl::OfferKeyEventL(const TKeyEvent&, TEventCode) |
| MediaGallery2.exe | CallThrdProcEntry |
| core.symbol | __ARM::default_unexpected_handler() drtaeabi.in(.text) |
| core.symbol <800fbd70> 000c | vtable for XLeaveException drtaeabi.in(.constdata__ZTV15XleaveException) |
| MGXUiBase.dll | CMGXAvkonViewImpl::OfferKeyEventL(const TKeyEvent&, TEventCode) |
| MGXListModel.dll | _E32Dll |
| MediaGallery2.exe | RunThread |
| MGXUiBase.dll | CMGXAvkonViewImpl::OfferKeyEventL(const TKeyEvent&, TEventCode) |

Table 5 Crash binary file originated from Nokia N95

The first table contains information of the trapped panic. The panicked module is Gallery, and panicked on one firmware release of Nokia N95. Other product specific data can be read from the table as well, which includes the IMEI code, the product type

and the ROM ID. System data that is related to the panic like stack base, stack pointer, program counter and available memory are also listed on the table. These system data can help to identify malfunctions and exceptions. For instance, continuous memory leak or heavy application loading can result in available memory lower than the usual boundary. Another example is that stack overflow can result in the stack pointer goes beyond the stack base.

The collected panic category and panic ID would be significant for pinpointing the software defect on code line, together with the dumped call stack. In many cases, panic is raised due to pass illegal parameters to library functions or misuse library functions. If such error is detected by the library code running on the same thread as the program and operating on behalf of the program, it will raise a panic and terminate the erroneous program. Panics are categorized by two attributes, the panic category and the panic ID. On the example crash binary file, the trapped panic has USER_EXEC as the category and 3 as the ID. Symbian OS Library provides reference to describe each panic type and programming error that leads to the panic. For the concerned panic on the example, Symbian panic reference explains like following:

"In Symbian OS 8.1b, 9.0, 9.1 and subsequent versions: this panic is raised when an exception is raised on the current thread by a call to *User::RaiseException()*, and the thread has no exception handler to handle the specified exception." [Symbian Panic Reference, 2006]

This points out that the suspicious code that may cause the panic is where a call to *User::RaiseException()* is placed. Developers can then go through the code of the panicked software module Gallery to locate the function call, and check whether there is any programming error. On the User Comment field, a brief description of how to reproduce the error is included. Developers may then debug the error by tracing the suspicious component while reproducing the erroneous behaviour on the device.

In some programs, developer may explicitly terminate the current thread if certain condition check fails. This is done by calling *User::Panic()* and passing a panic category and a panic ID as parameters. Developers can specify the category and ID to be any values, but not necessary Symbian OS defined values. For example, passing *KTestPanic* as the category and *5* as the ID when calling *User::Panic()* raises a Symbian user side panic. Mobile Crash will then trap the panic and generate a crash binary file, which has *KTestPanic* on the Panic Category field and *5* on the Panic ID field. Such crash binary files may help to directly pinpoint the error on code lines by searching code containing the customized panic category.

There are also situations that the panic category and ID are not collected. These may be caused by an unhandled exception instead of a panic. For these cases, developers would have to study the call stack to figure out the function calling sequence,

and then look for the same calling pattern from the panicked software component. In addition, User Comment and Test Set ID give hints of under what circumstance the panic happened, so to enable developers to reproduce the error.

The collected processor register sets are not listed on the example crash files. Usually they do not give much valuable information for solving the error. Among the register sets, R13 is the program counter and R15 is the stack pointer. These two registers are already listed on the basic crash information table. The collected run time binary modules and the address range where they are loaded to memory are not listed either. Those are mandatory for decoding call stack.

The complete call stack is much longer than what are presented on the second table. In most of the cases, developers may only be interested in the stack section under the stack pointer, which tells what are the uncompleted functions pushed onto stack. The stack presented on the example crash file includes only the section under the stack pointer, and only those stack words that can map to valid symbols. On the table, if a stack word is decoded from a software module map file (the module is included to ROFS image, and loaded to memory on demand), a row looks as following:

```
Adis.dll                CAdisImageProcessor::EventLoopL()
```

The stack word is mapped to function *CAdisImageProcessor::EventLoopL()* that is encapsulated to binary module *Adis.dll*. If a stack word is decoded from Core OS symbol file (Core OS is included to ROM image, and shadowed to RAM permanently), a row looks as following:

```
core.symbol
<8094695f> 0076      CFbsBitmap::SetScanLine(TDes8&, int) const      fbscli.in(.text)
```

The stack word is a ROM address, and lies within the memory range of *fbscli* component, which starts at address *8094695f* and its length is *0x76*. The stack word is mapped to function *CFbsBitmap::SetScanLine(TDes8&, int) const*.

[Symbian Panic Reference, 2006] Symbian Ltd, System panic reference, Available as http://www.symbian.com/developer/techlib/v9.1docs/doc_source/reference/N10352/index.html#PanicsReference%2eindex