Marika Salakari

# MEMORY MANAGEMENT IN EPOC

Final Year Project
Kajaani Polytechnic
Faculty of Engineering
Information Technology
Autumn 2000

**KAJAANIN AMMATTIKORKEAKOULU**

**INSINÖÖRITYÖ TIIVISTELMÄ**

| Osasto | Koulutusohjelma |
|---|---|
| Tekniikka | Tietotekniikka |

Tekijä(t)
Marika Salakari

Työn nimi
EPOCin muistinhallinta

| Vaihtoehtoiset ammattiopinnot | Ohjaaja(t) |
|---|---|
| Ohjelmistotekniikka | Heimo Kampman,<br>Harri Korhonen-Kosonen (NMP) |
| Aika<br>24.1.2001 | Sivumäärä<br>52 |

Tiivistelmä

Tämä lopputyö on tehty Nokia Matkapuhelimet Oy:lle. Työn tarkoituksena oli koota yhteen perustiedot liittyen EPOCin muistinhallintaan käytettäväksi opetusmateriaalina.

EPOC on Symbianin kehittämä ja hallinnoima käyttöjärjestelmä, joka on suunniteltu erityisesti pieniin kannettaviin laitteisiin, kuten matkapuhelimiin.

Muistinhallinta on yksi tärkeimmistä asioista suunniteltaessa sovelluksia kannettaviin laiteisiin, esimerkiksi matkapuhelimiin. Matkapuhelimissa on rajallinen määrä muistia ja matkapuhelin voi olla päällä yhtäjaksoisesti päiviä tai jopa kuukausia. Tästä syystä muistivuodot ovat hyvin kohtalokkaita ja voivat aiheuttaa paljon harmia ellei asiaa ole otettu huomioon jo suunnitteluvaiheessa.

Työssä käsitellään EPOCin perusrakennetta, muistia ja muistinhallintaan liittyviä asioita. Lisäksi käsitellään yhtä muistinhallinnan piirrettä, trap harnessia, tarkemmin. Lopuksi on katsaus testauksesta.

Luottamuksellinen
Kyllä
Ei X

Hakusanat
EPOC, Muistinhallinta

Säilytyspaikka
–

**KAJAANI POLYTECHNIC**

**ABSTRACT**
**FINAL YEAR PROJECT**

| Faculty | Faculty of Engineering | Degree programme | Information Technology |
|---|---|---|---|


Author(s)
Marika Salakari


Title
Memory Management in EPOC

| Optional professional studies | Instructor(s) / Supervisor(s) |
|---|---|
| Programming technology | Heimo Kampman,<br>Harri Korhonen-Kosonen (NMP) |

| Date | Total number of pages |
|---|---|
| 24 January 2001 | 52 |


Abstract

This final year project was done for Nokia Mobile Phones. The aim of the project was to congregate basic issues about EPOC Memory management. One part of EPOC memory management, called trap harness, is dealt with in more detail. Finally, testing tools suitable for testing in the EPOC environment are reviewed.

EPOC is a product of Symbian consisting of an operating system, core software, application frameworks, applications and development tools for WIDs.

Memory management is one of the most important items in applications which are designed for handheld devices, such as mobile phones. Mobile phones have limited memory and they run for days or months without rebooting. That is why memory leaks are very crucial and can cause considerable harm if the situation is not taken into consideration in the design phase.


Confidential
Yes
No X


Keywords
EPOC, Memory Management


Deposited at
–

PREFACE

This final year project was done for Nokia Mobile Phones Ltd., Oulu, Finland. The work for the project was carried out between June 2000 and January 2001.

I would like to thank Heimo Kampman, my supervisor in the project from Kajaani Polytechnic, for his guidance and constructive comments. I also would like to thank my Section Manager Sauli Hietakoste, who made the work possible and Harri Korhonen-Kosonen, my supervisor in the project from Nokia Mobile Phones, for advises and encouragement.

I would also like to say word of thanks to my colleagues in EPOC Applications section for their support and good working atmosphere, and all the other people within Nokia, who have helped me and given me valuable information for the project. I owe my warmest thanks to my son, to my parents and to my friends for the support they have given me throughout my studies.

Oulu, 24th January, 2001

Marika Salakari

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

GLOSSARY

| | |
|---|---|
| **API** | Application Programming Interface |
| **CONE** | CONtrol Environment |
| **DLL** | Dynamic Link Library |
| **EPOC32** | EPOC32 operating system |
| **GCC** | GNU C++ Compiler |
| **GUI** | Graphical User Interface |
| **HTML** | HyperText Markup Language |
| **MARM** | Multi-process ARM |
| **MMU** | Memory Management Unit |
| **OO** | Object Oriented |
| **OOM** | Out-Of-Memory |
| **OS** | Operating System |
| **PC** | Personal Computer |
| **PDE** | Page Directory Entry |
| **PIM** | Personal Information Management |
| **RAM** | Random Access Memory |
| **ROM** | Read Only Memory |
| **SDK** | Software Development Kit |
| **TCP/IP** | Transmission Control Protocol/Internet Protocol |
| **VGA** | Video Graphics Array |
| **WINC** | WINdows Connectivity |
| **WID** | Wireless Information Devices |
| **WINS** | WINdows Single process |

# 1 INTRODUCTION

Designers developing handheld devices face many challenges. There is limited amount of memory available in handheld devices, power management is important, and critical situations should be handled without losing user data, just to mention a few things that make the developing process more demanding than the developing process for a standard PC. The EPOC operating system was developed for the needs of wireless, handheld information devices (WIDs).

This thesis deals with the EPOC32 operating system and some special features in it, especially memory management. EPOC is Symbian's technology for mobile, ROM-based computing devices. Nokia has decided to use EPOC as the operating system in 3rd generation mobile phones. The first Nokia mobile phone with EPOC is Communicator 9210, which was introduced to the public some time ago.

The aim of this work was to gather basic issues about EPOC memory management. This document is suitable for acquainting new EPOC programmers with the basic principles of EPOC memory management.

First, some basic information about EPOC is given in section 1. Section 2 focuses on memory management, and in section 3 the basic issue is the trap harness. Testing and finding memory leaks are covered in section 4.

## 1.1 EPOC OS

EPOC is a fully 32-bit, ROM-based operating system developed, licensed and supported by Symbian. It was developed for the needs of wireless information devices, such as smartphones and communicators. [1]

EPOC's basic architecture is shown in figure 1. Core components provide APIs and a runtime environment on which all other components are built. A GUI and system components provide an environment and a programming framework for applications. The GUI provides standard controls and dialogs for programmers, a dialog framework, an application and object embedding framework, and many utilities. Communication components provide the APIs, drivers, link and higher-level protocols for a wide range of communication and data interchange requirements. Applications include the engines and GUIs, which directly implement end-user applications. [1] [2]

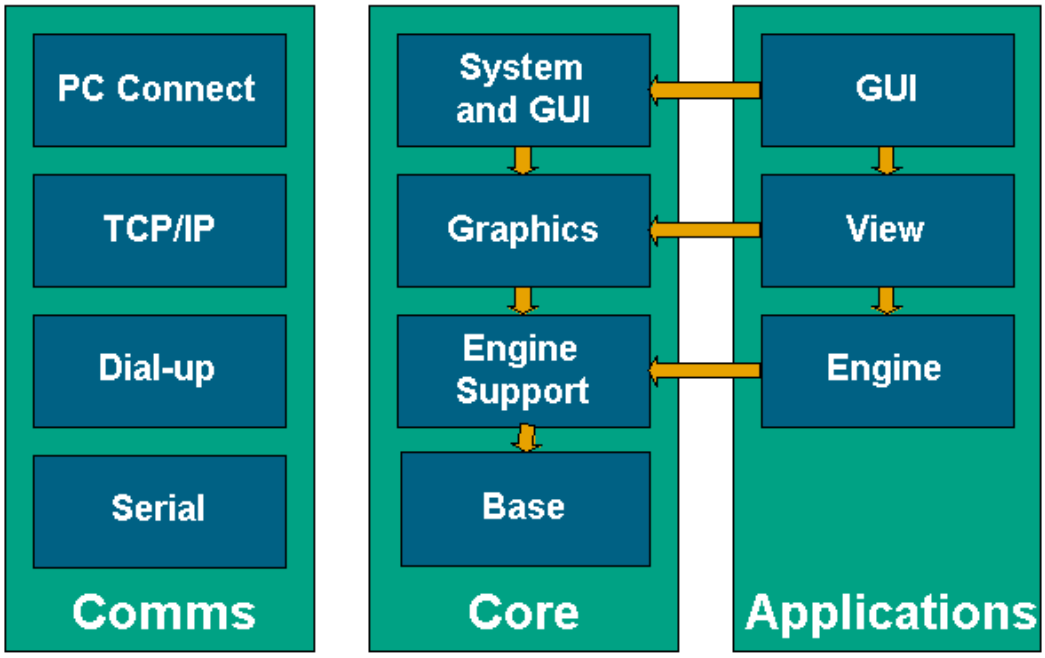


*Figure 1. EPOC's major components [2].*

One of the most important components in EPOC OS is the base. Figure 2 shows that the base may be divided into two parts: E32 and F32. E32 consists of a kernel and a user library, which provides classes, types, functions and services. F32 is a file server that provides file systems for ROM, RAM and removable media. F32 also provides an interface for dynamically installable file systems and an object-oriented client API for writing new file systems. [3] [4]

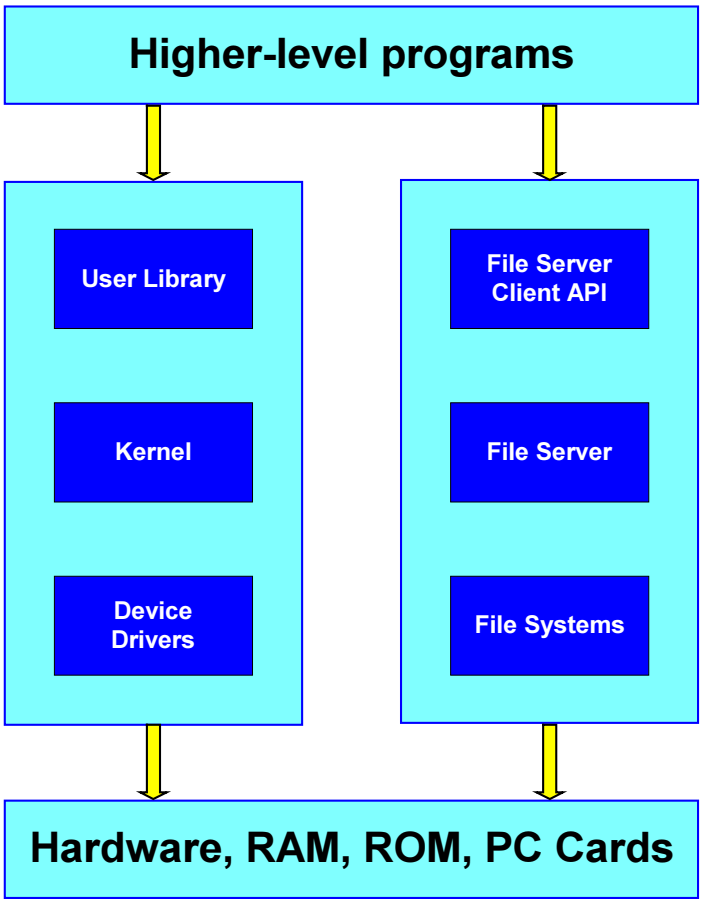


*Figure 2. Base components [5].*

The kernel manages the machine's hardware resources, such as system RAM and hardware devices. The kernel is fully privileged, so it can provide and control the way all other software components access resources. The user library, euser.dll, is the lowest-level user mode code, which offers library functions to other user-mode codes and controlled access to the kernel. [3]

In EPOC, the fundamental unit of protection is a process and the fundamental unit of execution is a thread. Each process has its own address space and each process has one or more threads. Threads run independently within the same address space, so it is possible that a thread can change memory belonging to another thread in the same process. [3]

EPOC is a portable runtime operating system with three major implementation families: target machines, a windows-based emulator and windows-based tools. EPOC target machines comprise a full operating system that boots on a ROM-based device and manages all aspects of that device – memory, scheduling, power, timers, files, keyboard, pointer, screen, PC cards, etc. The windows-based emulator makes it possible for the developer to run the software on Windows using Win32 services and the same user-side APIs as in the machine implementations. The windows-based tools are called a WINC environment. [6]

## 1.2 Background of the Symbian Consortium

Symbian is an independent joint venture formed by Ericsson, Nokia and Psion in June 1998. At the same time Motorola signed a Memorandum of Understanding to join Symbian, and in October 1998 Motorola became a shareholder in the Symbian joint venture. Matsushita (Panasonic) joined in May 1999. [3]

Symbian aims to promote standards for the interoperation of Wireless Information Devices, WIDs, with wireless networks, content services, messaging and enterprise-wide solutions. Symbian's mission is to license the Symbian platform to all mobile phone manufacturers and to create a mass market for next generation mobile phones by working closely with wireless networks, content, messaging and enterprise-wide solution providers. [6]

EPOC is Symbian's product, which consists of an operating system, core software, application frameworks, applications and development tools for WIDs. [6]

## 1.3 EPOC from the user's point of view

EPOC has outstanding reliability in all operating conditions. There are some primary requirements for devices containing EPOC. Power economy, requiring efficient applications, a small memory footprint, effective power management during normal operation, and robust response to low power conditions and sudden power supply failure. Because it is possible to make a connection using infrared, RS232 or sockets, and they have suitable higher-level application protocols, it is possible to integrate these devices with other wireless information devices, like portable computers. [7]

EPOC machines are expected to run for many days or weeks on batteries. The operating system and major applications may run for up to several years without ever being shut down or reset in any way. If an error occurs, it must be ensured that user data is still safe and will not disappear. EPOC has been designed with this in mind: ROM-based computing for low-power, compact machines and long-running, mission-critical applications. [7]

Several devices are available that have an EPOC operating system. Psion's Series 5mx was the first, and after that many others have already produced,such as Psion's Series 7 and Psion's Revo. [1] [3]

## 1.4 EPOC from the programmer's point of view

From the application developers viewpoint, the core components are EPOC's major components. They provide APIs and a runtime environment on which all other components are built (see figure 1). The core includes a base, engine support, graphics and a GUI. [1] [7]

The GUI and system components provide an environment and a programming framework for applications. EIKON is EPOC's GUI library, which has been designed specially for use with handheld devices that have a ½ VGA screen, a keyboard and a pen. EIKON provides standard controls and dialogs for programmers use, a dialog framework, an application and object embedding framework, and many utilities. [6]

EPOC32 operating system design is fully object-oriented and it is implemented in C++, except for a few hardware-dependent or extremely speed-critical functions written in assembly. [8]

Memory efficiency and cleanup are strengths in EPOC32. It is essential for an EPOC programmer to be familiar with the cleanup framework and to use it effectively. The programmer should be careful with allocating and deallocating memory, so that memory leaks do not occur. Memory management is one of the vital issues when developing applications in EPOC. More details about cleanup and memory management are given in section 2. [4]

Symbian does not provide a compiler of its own, but Visual C++ and GNU C++ compilers are used. The applications are developed in a Windows environment with Microsoft's graphical tools and debuggers, and then the same source code is recompiled using a GNU C++ compiler. After compiling, the source code can be transferred to the target machine using a link cable and suitable communications software. [4]

## 1.5 EPOC SDK

A software Development Kit (SDK) allows development of applications with a standard PC. The first version of the EPOC Software Development Kit (EPOC SDK 1.0) was released in August 1997. Symbian's first full release of EPOC is EPOC Release 5 (ER5) C++ SDK, which was

released in June 1999. In November 2000 Symbian launched its fully integrated, open software platform called v6.0. The issues handled in this chapter are based on version ER5. [3]

With EPOC Release 5 C++ SDK it is possible to write EPOC applications, DLLs and system components. The applications can be tested on a PC, since the SDK includes an EPOC simulator for Windows. [3] [9]

The SDK also includes documentation, an EPOC Emulator, a GCC cross compiler, various software tools, extensive example code, a complete design exercise and some selected EPOC source code. [4]

## 1.6 EPOC programming principles

EPOC uses C++, but it does not use some language features, it provides some new features of its own and it has adapted others. Most of these changes are made to achieve reliability or efficiency in ROM-based devices. [7]

### Basic classes

E32 applications use four general kinds of classes:

- Value classes, or types, whose name begins with a T. These may be safely orphaned on the stack because they need no explicit assignment or copy constructor operator.
- Heap-allocated classes, whose name begins with a C. These objects are derived from a CBase class, which means they have a constructor and a destructor, they are allocated on a heap, and they are usually referred to by pointer.
- Resource classes, whose name begins with an R. These classes are proxies for objects owned elsewhere.

- Mix-in classes, whose name begins with an M. These are the only classes where multiple inheritance is allowed. [7]

Naming conventions

EPOC has its own naming conventions for naming classes, structures, variables, functions, macros, enumerations, and constants. They reflect the cleanup-related properties of objects and classes. More details about this can be read from the book Professional Symbian Programming. [3] [7]

Recurring code patterns

There are some very commonly used code patterns. Two-phase construction is the most important to know, and it is covered later in this thesis.

Casting

EPOC32 provides some macros to encapsulate C++ cast operators. These macros are REINTERPRET_CAST, STATIC_CAST, CONST_CAST and MUTABLE_CAST. Casting should be used with caution, because it may indicate a questionable code. [7]

Asserts

Asserts are commonly used macros in EPOC. Asserts are one method of catching programming errors. Asserts are covered more precisely in section 4. [4]

### Test code

Test code is an important part of software development. Codes must be tested in specific areas, not only when the whole code is ready. Testing is covered more exactly in section 4. [7]

## 1.7 EPOC Programming Framework

Working with EPOC usually means working with EPOC frameworks. Frameworks are well suited to OO (object-oriented) systems, because frameworks take the notion of an API a step further and provide both architecture and ready-made building blocks for re-use. [4]

The programming framework is provided by EIKON. The EPOC32 programming framework provides the programmer with a large set of base classes and services required by all applications. For example, services like channelling, user input to the correct part of the application code, and providing access to file-handling functions defined by the Application Architecture. [4]

The programming framework requires that some aspects of an application's layout and behaviour, including menu, toolbar and other control structures, are defined in a resource file. This means that even small changes to the resource file can change an application significantly. This feature can be taken advantage of when localising an application. Only the resource file text associated with each menu item, task bar or other control needs to be changed. If the behaviour of the code does not change, it is not necessary to recompile the application to use the new file. [4]

# 2 MEMORY MANAGEMENT

## 2.1 General description

Dynamic memory management is a very important part of real-time systems. The amount of RAM in small, portable products is as small as possible. Because of costs and power consumption, that is why RAM must be shared among tasks.

There are some key issues to remember when developing software for wireless information devices. RAM is limited, so programs must use it efficiently. Because it is not possible to reserve unreleased resources, memory should be released as soon as possible. Out-of-memory situations will happen, so it is important to notice possible error situations beforehand. Every time memory is allocated, it is possibility to cause an out-of-memory error, and the operation could fail. If this happens all the allocated resources must be cleaned up and operation must roll back to an acceptable and consistent state without losing any user data. [7] [10]

Memory management is a way to control limited memory resources so that memory leaks do not happen. EPOC supplies a variety of tools for dealing with memory management. These tools include C++ destructor, a trap harness, a leave mechanism, a cleanup stack, heap failure tools, heap checking tools, a two-phase constructor pattern, a CBase class, some naming conventions and programming patterns. The cleanup stack, CBase class and two-phase construction are EPOC’s most important features. [4] [10]

## 2.2 Memory

There are three types of memory: program binaries, a stack and a heap. The program binaries are constant and do not change. Literals built by the programs go into the program binaries. The stack is the place where fixed-size objects whose lifetimes coincide with the function that creates them are placed. The heap is used for objects that are built or manipulated during runtime, are too big, or whose lifetimes do not coincide with the function that created them. [3]

### Process

Programs running in EPOC consist of a number of processes, which are memory management's basic units. Each process has its own address space, a primary thread, and any number of other threads. A primary thread is created every time a process is installed. Addition to the primary thread, processes may also create additional threads. [4]

Every user process has its own private address space. This address space is a collection of memory regions that a user process can access, but it does not include memory areas in the address space of another process. A kernel process is a special process whose threads run at a supervisor priviledge level. [4]

### Chunks

A chunk is a particular area in RAM, consisting of linear addresses next to each other. When a user process is created, it contains one thread and one to three chunks. A stack/heap chunk always exists. A code chunk exists if the process is loaded into RAM, and a data chunk exists if the process has static data. [4]

One chunk consists of two different kinds of regions, reserved and committed. A reserved region consists of linear addresses next to each other, which may be occupied by the chunk. The RAM is mapped in a committed region. Because the size of a chunk is dynamically alterable, the committed region's size can vary from zero up to the reserved region size. This gives some flexibility in case processes need to obtain more memory. [4]

The committed region generally starts from the bottom of the reserved region. It is also possible to create 'double-ended' chunks (from EPOC Release 4) where the committed region is any contiguous subset of the reserved region. The size of this kind of chunk is equal to an integral multiple of the processor page size. [4]

There can be two kinds of chunks:

- A local chunk, which is dedicated to the process which creates it. Other processes do not have access to it. Local chunks do not have names.
- A global chunk, which is intended to be accessed by other processes. These chunks have names by which a process can open them. [4]

Thread

A thread is a unit of execution within a process. Every process contains one primary thread. There may also be additional threads.

A kernel process is a special process, which normally contains two threads:

- A kernel server thread. This is the highest priority thread in the system.
- A null thread. When nothing else happens, this thread places the processor into an idle mode to save power.

Each thread is unaware of other threads in a process. Each thread also has also a priority. A thread with has highest priority, is executed first and

thread with higher priority can suspend executing thread with lower priority. Threads, which have same priority, are time sliced on a round robin basis. [4]

Each thread is assigned a priority; at any time, the thread running is the highest priority thread, which is ready to run. Context switching between threads involves saving and restoring the state of threads. This state includes not only the processor registers (the thread context) but also the address space accessible to the thread (the process context). The process context only needs switching if a reschedule is between threads in different processes. Active objects allow non pre-emptive multi-tasking within a single thread. [4]

After the thread is created it is in a suspended state, which means that the thread priority can be changed before the thread is started. The priority change has been done by calling the thread's member function called SetPriority(). The thread is started by calling thread's Resume() member function. [4]

Stack

T class objects are also called automatic variables. These T types can be declared on the stack without any kind of memory management. [7]

Below is an example code about T class object lifetime. Figure 3 illustrates this situation.

*Example 1. T-class object lifetime [7].*

```
{  // 1.
                TClass x;  // 2.
                 // use x
}  // 3.
```

The object's scope unit starts at point 1. In the figure it can be seen that the stack is empty. An object is created on the stack at point 2. After the object is used and the scope unit ends, at point 3, the object is automatically destroyed from the stack. [7]

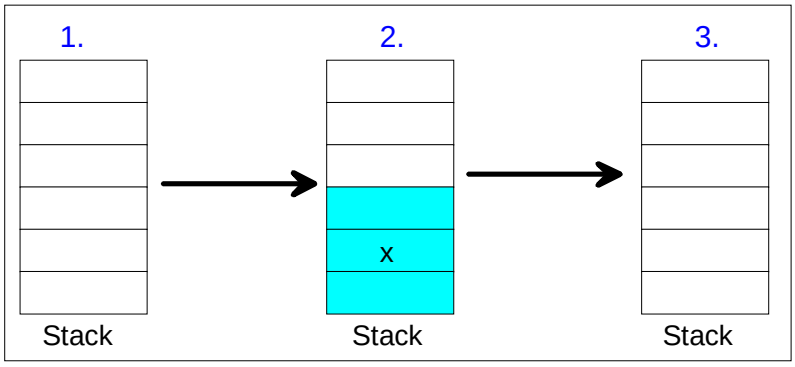


*Figure 3. T-class object lifetime.*

Heap

Each thread also has a heap. You can allocate and de-allocate objects on the heap at will, and refer to them using a pointer. The benefit of a heap is that the lifetime of an object is entirely within your control. This power comes with responsibility: you must not forget to de-allocate objects once you have finished with them, and you must not use pointers to objects that have been de-allocated. [4]

A heap's structure in EPOC is rather simple. It consists of two linked lists. One is for allocated cells and the another is for free cells. [7]

Object lifetime

Objects have a certain life cycle, which is the same in both a stack and a heap. The life cycle starts when memory is allocated for the object. Then this allocated memory is initialised, after which the object is used. After use all the resources used by that object are freed up. Finally, the memory is de-allocated from the stack or heap. [4]

Example 2 shows a normal C++ approach to object construction, apart from the use of an overloaded new operator. If the construction is successful, this approach behaves identically to that of C++ new. Figure 4 illustrates the use of memory resources. [7]

*Example 2. Object lifetime [7].*

```
{  // 1.
            CClass* p;  // 2.
            p = new( ELeave ) CClass; // 3.
              // Use the CClass
            delete p;  // 4.
}
```

New(ELeave) corresponds to new in C++, and it will leave if it fails to allocate the required memory. Operator new(ELeave) should always be used instead of plain new. [4]

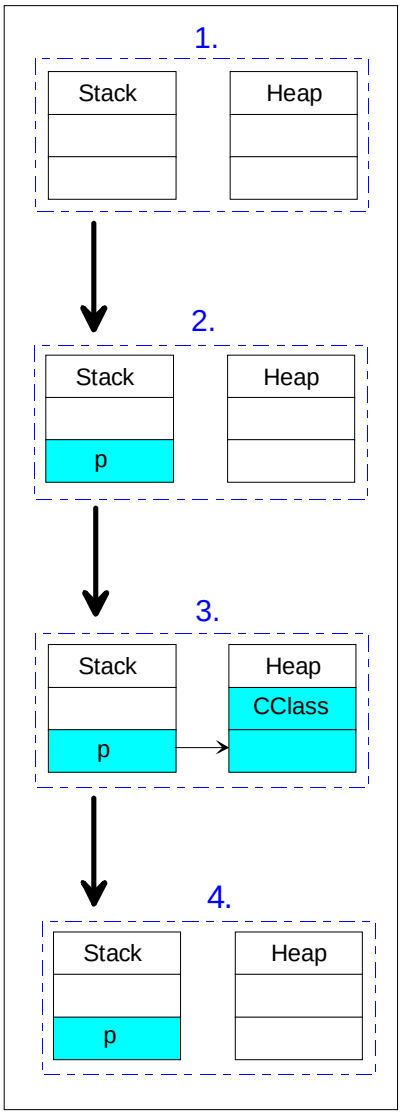


*Figure 4. Object lifetime*

From the figure above it can be seen that memory is allocated from the heap at point 3 and deallocated at point 4.

In some other operating systems memory is not as limited as in handheld devices, and that is why object lifetimes can be neglected. When a program terminates, the stack and heap are destroyed and there is no need to worry about cleaning up. But in EPOC, object lifetime is an important issue, because programs may run for months without interruption or a system re-boot. [4]

## 2.3 C++ exception handling

Exception handling mechanisms in C++ are provided to report and handle errors and exceptional events. The C++ mechanism try-catch works like EPOC's trap harness. Functions that can detect and recover from errors are executed within a try block. This try block calls other functions that are able to detect exceptions. Catch is an exception handler, which handles exceptions by their parameters. [11]

Then why not use C++ exception handling in EPOC? There are some reasons for that. First of all, the GNU C++ compiler, which is used for MARM implementation, does not support exception handling. Secondly, the code fragments used to clean up stack frames for throw processing can add considerably to code size. This is true even if such code fragments are only be generated if objects with destructors are allocated on the stack. In any case, this is forbidden in EPOC. Extra stack space and run-time overheads are reserved by the stack format to support C++ exception handling, even for functions that do not require cleanup fragments. Run-time type information support is needed when using C++ exceptions, and this carries its own overheads. Even if the overheads of C++ exceptions are low, they tend to increase. If C++ exception handling were used, part of EPOC exception support would also be needed. [4]

## 2.4 Panics

A panic is an action which stops the program from running. A panic is caused by programming errors, and only way to fix this is to correct the program. Asserts help detect programming errors. These errors are checked by the _ASSERT_DEBUG macro. [3] [4]

Panics are discovered by a library code, which operates on behalf of the program. There are functions available that panic a thread with an error. The User::Panic() function panics a thread if the operating library code is in a DLL running in the same thread as the program. When the library code is in a server executing on behalf of another program, the server should panic the client thread using RThread::Panic(). Panics are characterised by a category, which is a sixteen-character string, and a number, which identifies the specific cause. Some panics are raised in debug builds only. [4]

## 2.5 Memory Leaks

There are two kinds of memory leaks: static and dynamic. A static leak occurs when there is a different number of allocations and deallocations in a piece of code. This kind of leak will cause a panic when the application is closed. Dynamic leaks are caused by an error condition. [4]

When a new C class object is created, it is necessary to have an automatic pointer for it on the stack or a member pointer in another class. Memory leaks occur when pointer goes out of range before the object in the heap is deleted. When this kind of situation happens, the memory area where object is can not be used anymore. [7]

Example 3 is about a memory leak. It is a fragment of code in which an object is created and used, but not destroyed. The allocated memory is not released from the heap because the pointer of the heap object was lost for some reason. In figure 5 the same situation is presented visually.

*Example 3. Memory leak [7].*

```
{  // 1.
                CClass* p;  // 2.
                p = new( ELeave ) CClass; // 3.
                  // Use the CClass
                  // delete p;
}  // 4.
```

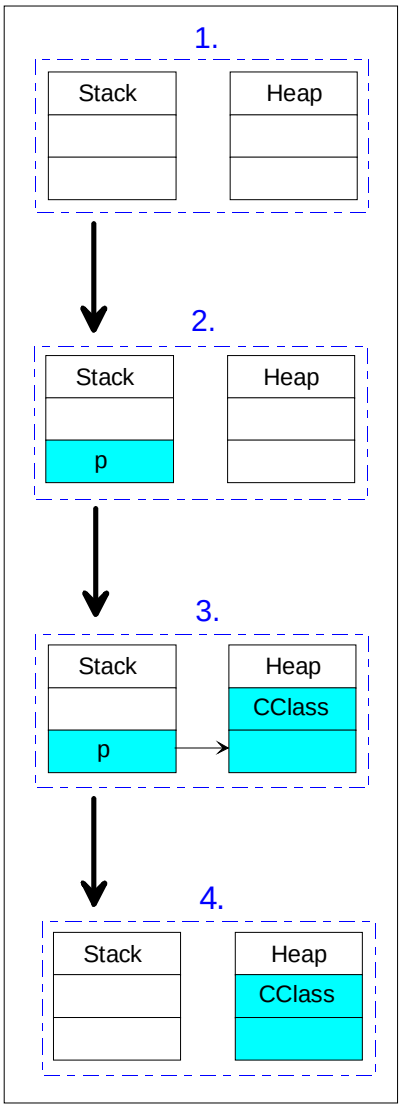


*Figure 5. Use of a stack and a heap when a memory leak happens.*

At point 3 the memory is allocated from the heap. If the memory is not deallocated from the heap before the pointer goes out of range, the memory can not be deallocated anymore and a memory leak occurs. The pointer of the heap object was lost and the object was orphaned and could not be cleaned up. This is a memory leak. [7]

Amusingly, a memory leak has been called ’alloc heaven’. This term comes from the fact, that an allocated heap cell is in heaven because it cannot be reached by pointers. Term ’alloc hell’ has also been suggested to describe the horror caused by double deletion, but that term never caught on. [3]

Memory leaks can be detected by the _UHEAP_MARK and _UHEAP_MARKEND macros. When memory is allocated, the number of allocated heap cells is noted by _UHEAP_MARK, and _UHEAP_MARKEND causes a panic if it finds that the number of heap cells currently allocated is not the same. These are very useful macros, which help to keep the heap imbalance in every development phase. [3] [7]

## 2.6 Cleanup Stack

When a function that has allocated objects leaves, there must be a way to clean up those objects. A cleanup stack is EPOC’s mechanism for handling this cleaning up. A cleanup stack must be used only when it is needed. Using a cleanup stack is quicker and more efficient than using a trap harness, so if appropriate, the cleanup stack should be used. The trap harness is explained in section 3. [4]

Every application has its own cleanup stack. Before it can be used, the cleanup stack must be created using CTrapCleanup::New(). All the

objects placed in the cleanup stack will be destroyed when a leave occurs. [4] [7] [12]

CleanupStack::PushL() pushes objects onto the cleanup stack and CleanupStack::Pop() cleans up the objects. Pop() can be used when it is sure that an object in the cleanup stack will either be destroyed or a reference to it will be stored. [4] [12]

Using the CleanupStack::PopAndDestroy() function will both pop and destroy the object from the cleanup stack. This is usually used when an object on the cleanup stack is no longer needed. [4]

Example 4 contains a fragment of code showing the use of the cleanup stack. At point A the object is pushed onto the cleanup stack, and after memory allocation has been completed, the object is popped from the cleanup stack.

*Example 4. Cleanup stack [7].*

```
CExample* CExample::NewL()
{
        CExample* self = new( ELeave ) CExample;
        CleanupStack::PushL( self );  // A
        self->iMem1 = new( ELeave ) CX;
        self->iMem2 = new( ELeave ) CY;
        CleanupStack::Pop();  // B
        return( self );
}
// delete self gets called by the cleanup stack if function leaves
```

In figure 6 the object is pushed onto the cleanup stack like at point A. If a leave occurs, the object is destroyed and the memory can be deallocated.

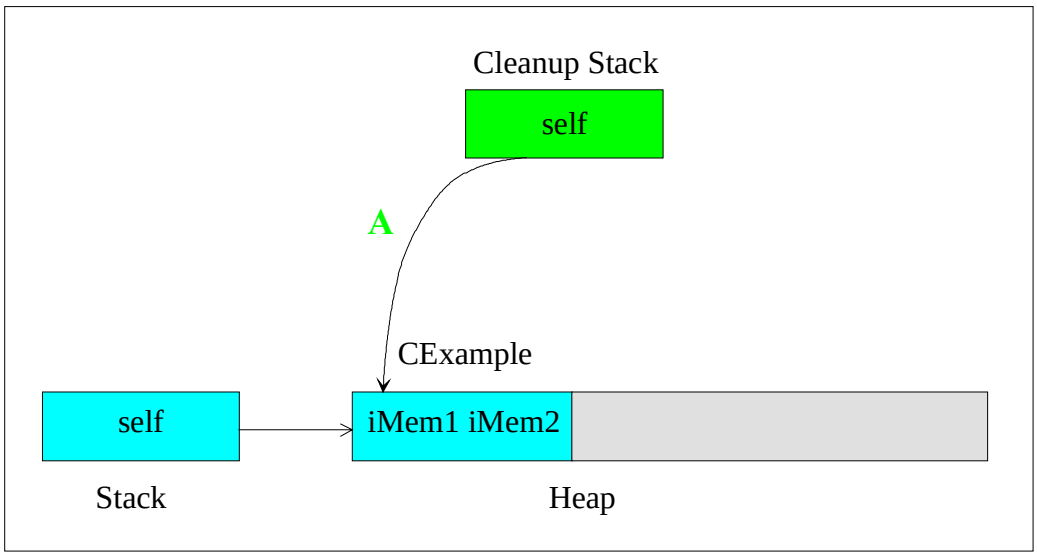


*Figure 6. Object pushed onto the cleanup stack.*

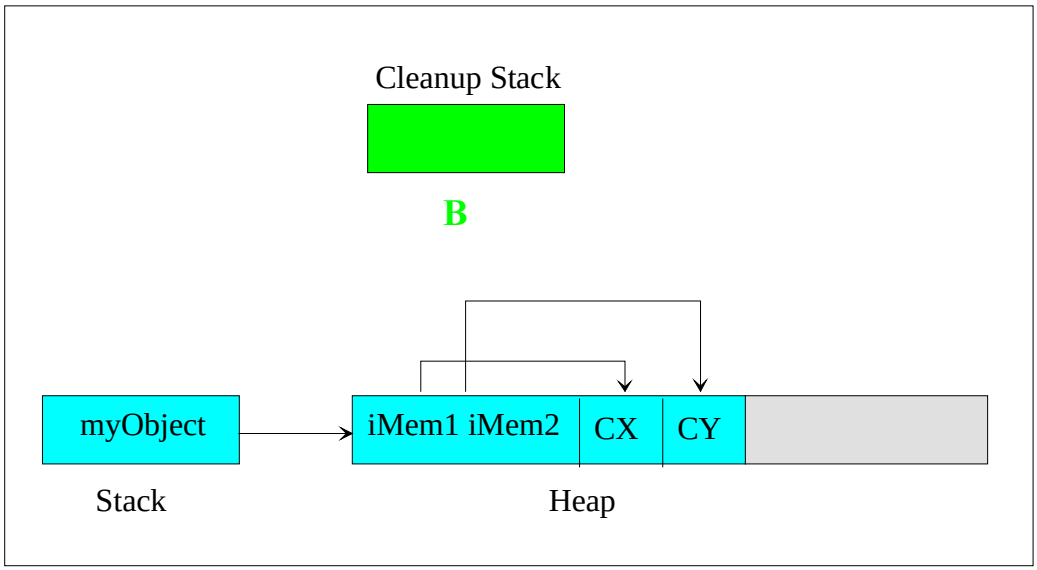


*Figure 7. Object popped from the cleanup stack.*

If the memory is succesfully allocated at point B, the object is popped from the cleanup stack, as shown in figure 7.

## 2.7 One-phase construction

It is appropriate to use a conventional C++ constructor when the object construction can not leave. A set of example codes follows, which presents two class definitions and a function that uses one-phase construction. One-phase construction uses the usual C++ construction strategy. [4] [7]

*Example 5. CSimple class definition [4].*

```
class CSimple : public CBase  // simple  class
{
                public:  // functions
                  CSimple();
                  ~CSimple();
                  void Display();
                private:
                  TInt iVal;
                protected:
                  CSimple( TInt );
};
```

All EPOC C classes are inherited from a base class called CBase, like the CSimple class in example 5 and CCompound in example 6. [4]

Example 6 is a definition of the CCompound class. The definition shows the data members and functions owned by the CCompound class. iChild is a pointer to a CSimple object owned by CCompound. The associations between the classes can also be seen in a class diagram in figure 8.

*Example 6. CCompound class definition [4].*

```
class CCompound : public CBase  // compound  class
{
                public:  // functions
                  void Display();
                  virtual ~CCompound();
```

```
                static CCompound* NewL();
                static CCompound* NewLC();
              private:
                CSimple* iChild;
               protected:
                CCompound();  // constructor
};
```

The object in example 7 is created using one-phase construction. In the example the PushL() and PopAndDestroy() functions are used to ensure that if a leave occurs the object can still be deleted correctly.

*Example 7. Function implemented using one-phase construction.*

```
void CExampleAppUi::OnePhaseL()
{
        CSimple* mySimple = new( ELeave ) CSimple;
        CleanupStack::PushL( mySimple );
        //
        // do something that might leave here
        //
        CleanupStack::PopAndDestroy();
        mySimple = 0;
}
```

The previous example also illustrates a good practise: zero the pointer after the object does not exist anymore.

Use of NewL() and NewLC()

The NewL() function is a static function that is usually used when the object being created is going to be referred to by an automatic variable. When creating member variables ( i front of the variable name), the destructor handles the cleanup, so NewL() is used. [7]

Example 8 contains the NewL() function. The object is popped from the cleanup stack before returning.

*Example 8. Heap pointer construction [7].*

```
CExample* CExample::NewL()
{
		Example* self = new( ELeave )CExample;
		CleanupStack::PushL( self );
		self->ConstructL();
		CleanupStack::Pop();
		return( self );
}
```

Example 9 uses the NewLC() function, which operates much like NewL(), but it does not pop the object from the cleanup stack before returning.

*Example 9. Stack pointer construction [7].*

```
CExample* CExample::NewLC()
{
		Example* self = new( ELeave )CExample;
		CleanupStack::PushL( self );  // A
		Self->ConstructL();
		return( self );
}

// Anywhere within another function
CExample* myEx = CExample::NewLC();
myEx->DoSomething();
CleanupStack::PopAndDestroy();
```

NewLC() is usually used to create objects owned by other objects. The pointer myEx must be deleted after use, or it will be orphaned in the stack. [7]

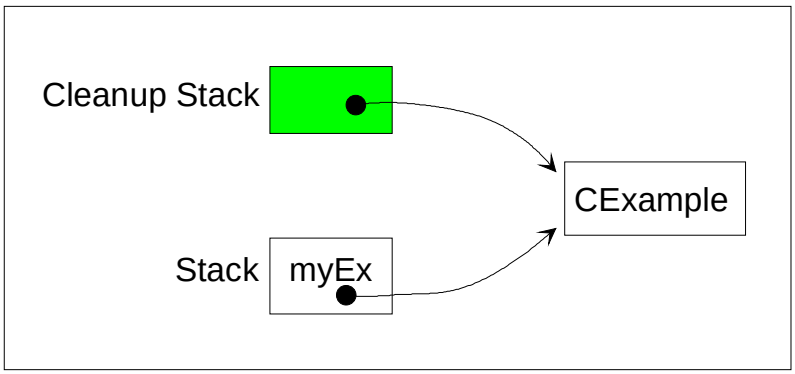


*Figure 8. Stack pointer construction.*

Figure 8 illustrates the use of a stack and a cleanup stack when creating a stack pointer. The  pointer is pushed onto the cleanup stack at point A, so if the construction fails there is no memory leak.

## 2.8 Two-phase construction

When using normal C++ conventions between the allocation performed by new and the invocation of a C++ constructor that follows the allocation, there is no need to push objects onto the cleanup stack. The C++ constructors should contain any functions that can leave. That is why two-phase construction was invented. Two-phase construction should be used when a class has to allocate member data storage, in other words, for C classes. [3]

The next code example uses classes like the example presented earlier in one-phase construction, but now a two-phase constructor is used.

*Example 10. Function implemented using two-phase construction.*

```
void CExampleAppUi::TwoPhaseL()
{
		CCompound* my2Phase = CCompound::NewLC();
		//
		// do something that might leave here
```

```
        //
        CleanupStack::PopAndDestroy();
}
```

After the construction is successfully finished, the pointer is popped and destroyed from the cleanup stack.

*Example 11. NewLC() function.*

```
CCompound* CCompound::NewLC()
{
        CCompound* self = new ( ELeave ) CCompound;  // A
        CleanupStack::PushL( self );
        self->ConstructL();  // B
        return self;
}
```

After allocating memory from the heap at point A, the pointer is pushed onto the cleanup stack. If the function leaves, the pointer is safe and the memory can be deallocated. After pushing the pointer onto the cleanup stack, the second-phase constructor is called at point B, and the construction can be finished. [7]

Example 12 contains the ConstructL() function, which is called the second-phase constructor. This function performs any initialization that might leave. [7]

*Example 12. CCompound's constructor in two-phase construction [4].*

```
void CCompound::CConstructL( )
{
        iChild = new( ELeave ) CSimple;
}
```

When performing initialization that could leave, a two-phase constructor must be used. That is a reliable way to avoid leaves when initializing classes.

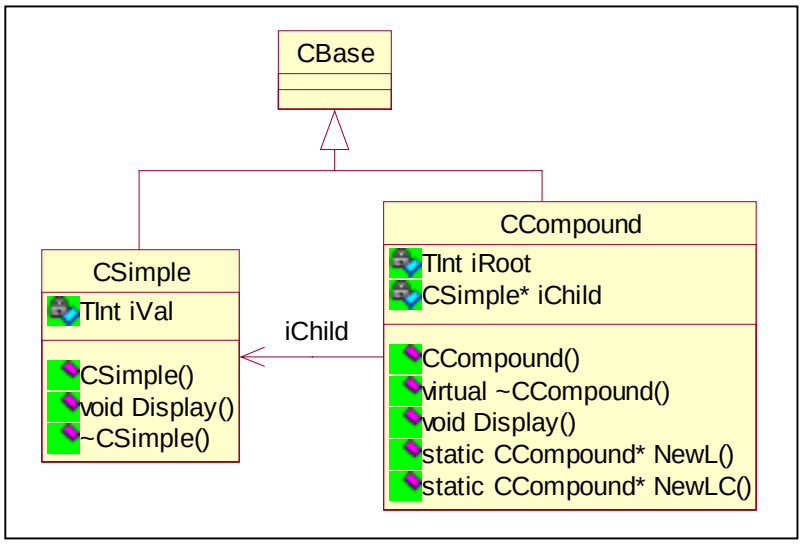


*Figure 9. Class diagram.*

The associations between the CSimple, CCompound and CBase classes can be seen in figure 8. From the diagram it can be seen that the CCompound class owns the other objects. iChild points to a CSimple object owned by CCompound.

## 2.9 General rules for cleanup

It is important to follow certain instructions when handling the cleanup. Very few rules govern cleanup stack programming, and they are relatively easy to learn. Here are some basic instructions that help in handling the cleanup correctly. Most of these instructions are based on EPOC Release 5 C++ SDK, System Documentation HTML Help [4] and the book Professional Symbian programming [3].

When allocating objects, it is necessary to ensure that if a leave occurs, the object will not disappear. So, after the object is allocated and before a pointer of that object is stored in a structure that would be accessible if a leave occurs, the object must be pushed onto the cleanup stack. After the pointer is stored in an object that is accessible after a possible leave, the object should be popped from the cleanup stack. [4] [13]

The cleanup stack is needed only when it is necessary to prevent the object's destructor from being bypassed. If the object's destructor is going to be called anyway, or if the object is a member variable of another class, the cleanup stack is not needed. A member variable should never be pushed onto the cleanup stack. [10] [13]

Never push member variables onto the cleanup stack. If this is done it is likely that the member variables will be deleted from both the cleanup stack and the class's destructor. [3] [13]

# 3 TRAP HARNESS

## 3.1 Purpose of the trap harness

A trap harness is a Symbian coding convention that works basically like C++'s and Java's try-catch mechanism. When a leave occurs, it means that either a resource allocation has failed or a request for the use of some remote resources has not been successful. [7]

## 3.2 Leave functions

When an OOM error occurs, functions in EPOC32 should leave. A function may also leave because a function it called has left. When a leave occurs, the User::Leave() function is invoked and the program will return to the current trap harness. [3] [13]

It is important to know if a function might leave. When writing this kind of function, L should be placed in the function's name. However, the compiler does not check leave function names, so if the L is missing, the compiler will not complain about it. [3] [13]

When writing a function it must be known if the function is an L function or not. Every time when it is possible that a function might require a cleanup, the function must be an L function. The function must also be an L function if the function calls another function, which is an L function. [3]

So, a trap harness invokes only L functions, but that does not mean all L functions are invoked by a trap harness. If it were so, exception handling would be too inefficient to use. [4]

## 3.3 Leave Mechanism

When an OOM error occurs, the operation must roll back to a former state that was consistent and stabile. A mechanism that makes this possible is called a trap harness. A trap harness catches any function that leaves inside the trap harness. When a function leaves, the TRAP or TRAPD macro takes control and returns an error code, which can be used by the calling function. [3] [4]

*Example 13. Using TRAPD [7].*

```
TRAPD( error, doExampleL() );
  if( error != KerrNone )
  {
                //do some error code
  }
```

Example 13 above shows the syntax required when a TRAPD macro is used. The doExampleL() function is executed under a trap harness, and if the function leaves, the error code inside the TRAPD is executed.

The only difference between the TRAP and TRAPD macros is, that in TRAP the program code must declare the leave code variable. TRAPD is more convenient to use as it declares this inside the macro. So, when TRAP is used like TRAPD is used in example 13, the code looks like the one shown in example 14. [13]

*Example 14. Using TRAP [7].*

```
TInt error;
TRAP( error, doExampleL() );
  if ( error != KerrNone )
  {
                //do some error code
  }
```

When a leave occurs, the call stack is unwound back to the last TRAP and the cleanup stack is unwound, too. All EPOC code is TRAPed in the active scheduler of the thread. Default TRAP behaviour displays trivial "Error" dialogs. When a function leaves under a trap harness, the TRAP macro takes the control and stores the leave code into the TRAP's return value. [4]

*Example 15. TRAP example [4].*

```
TInt leaveCode;  // hold result from trap
TRAP( leaveCode, SomeFunctionL() );  // call a function
   if( leaveCode != KerrNone )  // check for error leave code
   {
            //do something
   }
```

In the previous example the function called by TRAP is executed under a trap harness. If the function leaves, control returns immediately to the TRAP macro.

It is also possible that the called function returns a value. The syntax for this action is shown in example 16.

*Example 16. Return of a value [4].*

```
TRAPD( leaveCode,value = GetSomethingL() ); // get a value
   if( leaveCode != KerrNone ) // check for error leave code
   {
            // some cleanup
   }
   else {
            // didn't leave: value valid
   }
```

## 3.4 Nested Traps

Traps inside of other traps are needed sometimes. A situation like this appears when a new trap harness is set up by a function executing within an existing trap harness. If the function leaves, control is transfered to the new trap harness, only. [3] [4]

Lets take an example, like a word processor. When a key is pressed it causes many actions, like allocation of undo buffers and expansion of the document to receive a new character. If anything goes wrong the whole operation need to be undone completely. Example 17 contains code which can be used in a situation like the one described above. [3]

*Example 17. Specialised cleanup [3].*

```
TRAPD( error, HandleKeyL() );
if( error )
{
   RevertUndoBuffer();

   //Any other special cleanup
   User::Leave( error );
}
```

This performs a specialised cleanup and then leaves anyway, so the EIKON framework can post an error message. More is mentioned about error messages in chapter 3.5.

Some Draw() functions are rather complicated, and it may be appropriate to code them in such a way that they make allocations. In this case, the failures are trapped, as shown in example 18. [3]

*Example 18. Example of a Draw() function [3].*

```
virtual void Draw( const Trect& aRect ) const
{
   TRAPD( error, MyPrivateDrawL(aRect) );
}
```

Example 19 shows a useless trap code.

*Example 19. Useless trap code [3].*

```
TRAPD( error, FooL() );
if( error )
   User::Leave( error );
```

The previous code does exactly the same as the code in example 20.

*Example 20. Substitute function [3].*

```
FooL();
```

From these two examples above it can be seen that it is not wise to use a trap in every situation. Traps increase the code size and take time to execute. That is why traps should be used only when they are needed.

## 3.5 Error notification

There are three different kind of errors:

- program errors, which are caused by the programmer.
- environment errors, which are caused by insufficient memory or disk space, or other missing resources

- user errors, which are caused by the user, for example by attempting to enter bad data into a dialogbox. [4]

When a function leaves, some kind of error indication should be performed to indicate that the function was not actually executed. The next two examples, 21 and 22, present two different ways to notify of an error. Example 21 uses a subsequent leave to notify of an error:

*Example 21. Error notification by a subsequent leave [4].*

```
void TrySomethingL()
        {
        TRAPD( leaveCode, SomeFunctionL() );
         if( leaveCode )
                {
                //cleanup
                User::Leave( KerrSomethingDrastic );  //leave because of
                                                      //error
                }
        }
```

So, if an error situation occurs, the function leaves.

In example 22 the error is announced by an error code. Different kind of errors cause different kinds of error codes. Error code value can be used to inform the user which kind of error happened.

*Example 22. Error notification by an error code [4].*

```
TInt TrySomething()
    {
    TRAPD( leaveCode, SomeFunctionL() );
    if( leaveCode )
        {
        //cleanup
        return KerrSomethingDrastic;
            // indicate couldn't do request function
        }
    return KerrNone;  // indicate requested function performed
    }
```

If the previous SomeFunction() function leaves, an error code is returned. If the function does not leave, it returns a KerrNone to indicate that the function was performed succesfully.

In some circumstances, it is necessary to use no error notification. For example, if User::Leave() prevents an action from occurring and the cleanup stack takes care of all the cleanup requirements, it is appropriate to leave with a reason code KerrNone. [4]

## 3.6 Use of the trap harness

Different error conditions are checked and handled in different ways. Program errors are checked by the _ASSERT_DEBUG macro. These errors cause a panic, i.e., program running is terminated. Environment and user errors can be handled in two ways:

- If an error is detected before an action is performed, an error message is a convenient way to notify about it. This checking and dealing with a return code is usually more economical than setting up a trap harness. This is also quite easy to program, and if cleanup requirements exists they are also easy to identify or handle.

- If an error can not be detected until the processing of a requested action, the process should be run under a trap harness. The error can be signalled by User::Leave(), so that the cleanup stack handles the cleanup and the code follows the trap harness. When an error occurs deep inside the processing of a requested action, the trap harness is the more appropriate method to use. [4]

The situation may be more complicated when interfaces exist. Then there may be a situation in which both of the methods, or even a combination of both methods, is used. This kind of situation exists when an L function has to be used: it is necessary to use a trap harness or ensure that the program is already running in one. All CONE applications run under a CONE-provided trap harness. When specifying interfaces for others to use, there can be a significant difference between which method is used, leave or error return. It will influence the design of the programs that use the interface. [4]

# 4 CODE TESTING

## 4.1 Purpose of code testing

Code testing is an important part of the software development process. When developing applications like software for a mobile phone, it is important to carry out low memory testing from time to time. For debug builds, EPOC provides heap failure tools.

## 4.2 Testing tools

Application programmers mostly use the resources consisting of the memory in their own application. EPOC's toolkit has its own cleanup tools for handling and testing out-of-memory. Part of these tools have been covered earlier in this thesis. [3]

EPOC has own debugging tools for searching for memory leaks. These tools are called EIKON debugging tools, and they can be used when running a program with EPOC's own debugger. Some memory leaks can be eliminated by using these tools. After building and translating the code for the target machine, other tools must be used. [3]

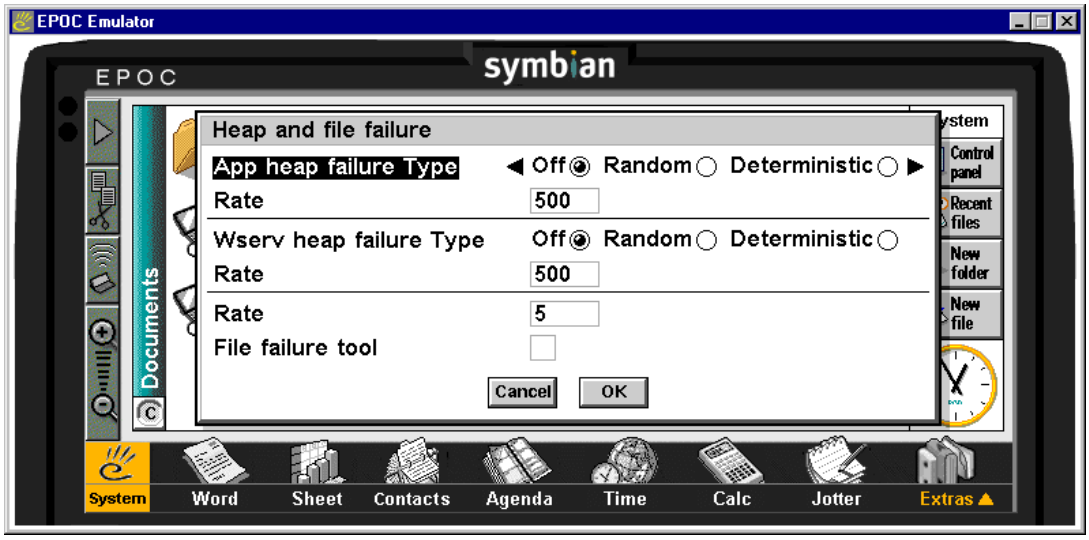


*Figure 10. EPOC heap failure tool [13].*

Heap failure tools, which produce deliberate out-of-memory errors are created for testing purposes. When ctrl+alt+shift+P is pressed, the heap failure dialog shown in figure 3 appears. There are three options:

- off: if there is sufficient memory, the allocations succeed
- random: if there is sufficient memory, the attempted allocations usually succeed, but randomly fail
- deterministic: depending on the number typed in the Rate field, the allocations are guaranteed to fail. For example, if the rate is 5, the allocations will fail on the 5th, 10th etc., attempts. [3] [7]

### 4.2.1 EPOC SDK testing tools

EPOC's emulator is a testing and developing tool which can be used to find possible memory leaks. It is used alongside any language-specific tools provided by the development environments. [4]

EIKON GUI provides special key combinations for resource checking and redraw testing. These can be used in debug builds of EPOC to help trap memory leaks early in the development cycle. EIKON debugging keys for resource allocation are:

- ctrl+alt+shift+A; shows the number of heap cells allocated by the program.
- ctrl+alt+shift+B; show the number of file server resources used by this program.
- ctrl+alt+shift+C; shows the number of window server resources used by this program. [4]

WINS (Windows single process) emulator is a good help when solving memory leaks. When an application is closed, the emulator will panic and raise a dialogbox if memory has leaked. This dialogbox gives the address of the leaked memory, which is why you should exit the application, not just kill the emulator. An application should always exit cleanly, even in the development phase. [7]

The next thing is to find what leaked. If the application exit panics, use the watch window in Visual C++ to try to cast the leaked address to CBase*. This works only if the leaked object is CBase derived. Once the address of the leaked memory is known, you can find where it was allocated. [3]

These techniques are not foolproof. If a large compound object has leaked, you may get bogus type information. The same address may be allocated many times, so you do not know which one is the leak. Both techniques rely on the leak being repeatable. Leaked resources can be very difficult to find unless the server panics. A better way to eliminate memory leaks is avoidance. [3]

Memory leaks can be found by using _UHEAP_MARK and _UHEAP_MARKEND in the code to check for mismatched heap allocs and frees or by using _K variations for the kernel heap (device drivers, etc.). [3] [7]

### 4.2.2 Commercial testing tools

There are not so many commercial testing tools available on the market, which are suitable for the EPOC environment. Some testing tools, like McCabe IQ and TestWell Toolpack++, support EPOC code testing in the PC Emulator environment, but every one of these testing tools has problems with the EPOC environment. So, modifications must be made to get the tools to work properly. A study is going on, which is examining the suitability of different testing tools for this purpose.

# 5 SUMMARY

Memory resources in handheld devices, are limited and if resourses are allocated but never freed, this will cause memory leaks and out-of-memory situations. When programming with EPOC it is important to know the basic issues about memory management and cleanup.

This document is suitable for new EPOC programmers to allow them to get acquainted with the basic principles of EPOC memory management. The trap harness is one of the specific features of EPOC, and this is covered more precisely in this work. Because there is also other solutions, the trap harness is a little-used feature in memory management, but in some cases it is the only way to handle situations like error conditions.

EPOC was new to me when I started this project. First I started to search for and read material about EPOC and its memory management. At the moment, all available material has been produced by Symbian, so this thesis work is almost completely based on Symbian’s materials.

A good thing was that during this final year project I learned much about EPOC. I believe this work has taught me many things that will be helpful in my further work. With the help of this work new employees and students can get familiar with memory management in EPOC.

Further development

Because the contribution of memory management is one of the most important issues when developing applications for handheld devices, the importance of testing is very obvious. The next thing to do would be to look for suitable testing tools, which support testing in the EPOC environment. Especially the possibility of testing memory leaks during the implementation phase.

# 6 REFERENCES


1 Symbian. EPOC Overview: Summary. Last modified 23. May 2000. [WWW-document]. <http://www.symbian.com/technology/papers/e5oall/e5oall.html>

2 Symbian. EPOC Overview: Core. Last modified 23. May 2000. [WWW-document]. <http://www.symbian.com/technology/papers/e5ocore/e5ocore.html>

3 Tasker, M. Professional Symbian Programming. Wrox Press Ltd. ISBN 1-861003-03-X.

4 EPOC Release 5 C++ SDK, System Documentation HTML Help

5 Tasker, Martin. Symbian. EPOC Overview. January 1999.

6 Symbian. About us: Corporate Fact Sheet. Last modified 25. May 2000. [WWW-document]. <http://www.symbian.com/about/corpfacts.html>

7 Tieturi. EPOC Programing Essentials. Course material. 2000.

8 Symbian. Technical Library. C++ Development process. Last modified 04. February 2000. [WWW-document]. <http://www.epocworld.com/techlibrary/technotes/C++process.html>

9 Symbian. Getting started: About this SDK. Last modified 04. February 2000. [WWW-document]. <http://www.epocworld.com/techlibrary/documentation/ER5/CPP/sysdoc/product/productdocs/aboutcpp.html>

10 Symbian. Approaches to memory management. Last modified 23. May 2000. [WWW-document]. <http://www.symbian.com/technology/papers/memmanc/memmanc.html>

11 Stevens, Walnum. Standard C++ Bible. IDG Books Worldwide, Inc. ISBN 0-7645-4654-6

12 Symbian. Exeption handling in EPOC. Last modified 04. February 2000 [WWW-document]. <http://www.symbiandevnet.com/techlib/techcomms/techpapers/papers/cpp_tutorial_excephandling/excepthandling.htm>

13 Symbian. Memory management and cleanup. Last modified 06. November 2000 [WWW-document]. <http://www.symbiandevnet.com/techlibrary/techcomms/techpapers/papers/memman/memman.htm>