Coding Idioms for Symbian OS

1. Introduction

1.1 Purpose and scope

This document is aimed at all Symbian OS smartphone developers. The apocryphal 80-20 rule states that 80% of the time will be spent fixing 20% of the problems that occur in development. The aim of this document is to address those 20%.

2. Memory

This section contains a review of the techniques that Symbian OS provides for preventing memory related problems. All developers should understand this well – it is the holy grail of Symbian OS programming!

2.1 Review of the cleanupstack

2.1.1 All programs should check for out-of-resource errors

In any application, an error may occur at run time due to a lack of resources: for example, the machine running out of memory, or a communication port being unavailable. This type of error is known as an exception.

An exception should be distinguished from a programming error: a programming error can be addressed by fixing the program, but it is impossible to make a program free from the possibility of an exception occurring.

Therefore, programs should be able to recover from exceptions when they occur. This is particularly important in Symbian OS, for the following reasons:

- Symbian OS applications are designed to run for long periods (months or even years) without interruption or system re-boot.
- Symbian OS applications are designed to run on machines that have limited resources, particularly memory. Therefore, an out-of-resource error is more likely to occur than in a desktop application.

2.1.2 Conventional error-checking methods

In a conventional C or C++ program an if statement might typically be used to check for an out-of-resource error. For example:

```
if ((myObject = new CSomeObject()) == NULL)
           PerformSomeErrorCode();
```

2.1.3 Problems with conventional methods

There are two problems with this conventional approach:

1. It requires many extra lines of code to put such error checking round every single function that might cause an out-of-resource error. This increases code size and reduces readability.
2. If a constructor fails to allocate resources, there is no way to return an error code, because constructors have no return value. The result would be an incompletely allocated object, which could cause a program crash.

C++ exception handling (try, catch and throw) provides solutions to these problems, but it is not used in Symbian OS, because of its large overheads in code size. Instead, Symbian OS provides its own system of exception handling.

2.1.4 Symbian OS's solutions

Symbian OS applications can achieve efficient exception handling by following the rules below:

Rule 1: All functions that can leave should have an 'L' on the end of their name. Leaves propagate back up the call stack though functions named with 'L' until they are caught by a trap harness, which is normally provided by your thread's Active Scheduler.

Rule 2: If you allocate memory on the heap, and the pointer to it is an automatic variable (i.e. not a member variable), it should be pushed to the cleanup stack so that it can be deleted when a leave occurs. All objects pushed to the cleanup stack should be popped before they are destroyed.

Rule 3: A C++ constructor or destructor must not leave or fail. Therefore, if construction of an object can fail with an out-of-resource error, all the instructions that might fail should be taken out of the C++ constructor and put in a ConstructL() function, which should be called after the C++ constructor has completed.

2.2 RULE 1: FUNCTIONS THAT LEAVE, AND TRAP HARNESSES

2.2.1 Functions that leave

Instead of returning an error code, functions in Symbian OS should leave whenever an out-of-resource error occurs. A leave is a call to User::Leave(), and it causes program execution to return immediately to the trap harness within which the function was executed.

All functions that can leave should have the suffix "L". This enables programmers to know that the function might leave.

For example,

```
Void MyFunctionL()
           {
           iMember = new (ELeave) CMember;
           iValue = AnotherFunctionL();
           User::LeaveIfError(iSession.Connect());
           }
```

Each line in MyFunctionL could cause a Leave, this is why MyFunctionL is a leaving function.

Note however that it is VERY RARELY necessary for application code to use a trap, since the application framework provides traps in the right places, and code to handle them. You should NOT use traps in normal coding. In general the way to deal with Leaves is to simply allow them to propagate through your function by adding an L to the function name.

2.2.2 new (ELeave)

The possibility of new failing arises so often that in Symbian OS the new operator has been overridden to take a parameter, ELeave. When called with this parameter, the overridden new operator will leave if it fails to allocate the required memory. This is implemented globally, so any class can use new (ELeave).

So,

```
CSomeObject* myObject = new CSomeObject;
if (!myObject) User::Leave(KErrNoMemory);
```

can be replaced by

```
CSomeObject* myObject = new (ELeave) CSomeObject;
```

in Symbian OS programs.

2.2.3 The NewL() and NewLC() conventions

By convention, Symbian OS classes often implement the functions NewL() and NewLC().

NewL() creates the object on the heap and leaves if an out-of-memory error occurs.

For simple objects this simply involves a call to new (ELeave). However, for compound objects it can incorporate two-phase construction: see "Rule 3" below.

NewLC() creates the object on the heap, leaving if an out-of-memory error occurs, and pushes the object to the cleanup stack: see "Rule 2" below.

In general, when creating C-class objects, programs should use `NewL()` if a member variable will point to the object, and `NewLC()` if an automatic variable will point to it.

However, it is not always advisable to implement `NewL()` and `NewLC()` for every class. If NewL() or `NewLC()` are called from only one place in the application, implementing them will actually use more lines of code than it saves. It is a good idea to assess the need for `NewL()` and `NewLC()` for each individual class.

Note: `NewL()` and `NewLC()` must be declared as static functions in the class definition. This allows them to be called before an instance of the class exists. They are invoked using the class name. For example:

```
CSomeObject* myObject=CSomeObject::NewL();
```

2.2.4 Using a trap harness: TRAP and TRAPD

In exceptional circumstances it is permitted to handle a leave yourself by using a TRAP. However, the use of TRAP and TRAPD is only for special situations – for all general coding they should not be used. Usually the best course of action is to allow the Leave to propagate back to the Active Scheduler for default handling. If in any doubt about whether you really need to use a TRAP, there is probably a cheaper or cleaner way to achieve the same functionality.

Symbian OS provides two trap harness macros, TRAP and TRAPD, which are both very similar. Whenever a leave occurs within code executed inside the harness, program control returns immediately to the trap harness macro. The macro then returns an error code which can be used by the calling function.

To execute a function within a trap harness, use TRAPD as follows:

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

TRAP differs from TRAPD only in that the program code must declare the leave code variable. TRAPD is more convenient to use as it declares it inside the macro. So, using TRAP, the above example would become:

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

Any functions called by `doExampleL()` are also executed within the trap harness, as are any functions called by them, and so on: a leave occurring in any function nested within `doExampleL()` will return to this trap harness. Another trap harness can be used within the first, so that errors are checked at different levels within the application.

2.3 RULE 2: USING A CLEANUP STACK

If a function leaves, control returns immediately to the trap harness within which it was called, generally the default TRAP inside the thread’s Active Scheduler. This means that any automatic variables within functions called inside the trap harness are destroyed. A problem arises if any of these automatic variables are pointers to objects allocated on the heap: when the leave occurs and the pointer is destroyed, the object it pointed to is orphaned and a memory leak occurs.

Example:

```
void doExampleL()
{
CSomeObject* myObject1=new (ELeave) CSomeObject;
CSomeObject* myObject2=new (ELeave) CSomeObject;           // WRONG
}
```

In this example, if `myObject1` was `new`'ed successfully, but there was insufficient memory to allocate `myObject2`, `myObject1` would be orphaned on the heap.

Thus, we need some mechanism for retaining any such pointers, so that the memory they point to can be freed after a leave. Symbian OS's mechanism for this is the cleanup stack.

2.3.1 Using a cleanup stack

The cleanup stack is a stack containing pointers to all the objects that need to be freed when a leave occurs. This means all C-class objects pointed to by automatic variables rather than instance data.

When a leave occurs, the TRAP or TRAPD macro pops and destroys everything on the cleanup stack that was pushed to it since the beginning of the trap.

All applications have their own cleanup stack, which they must create. (In an Series 60 application, this is done for you by the Series 60 framework.) Typically, all programs will have at least one object to push to the cleanup stack.

Objects are pushed to the cleanup stack using `CleanupStack::PushL()` and popped using `CleanupStack::Pop()`. Objects on the cleanup stack should be popped when there is no longer a chance that they could be orphaned by a leave, which is usually just before they are deleted. `PopAndDestroy()` is normally used instead of `Pop()`, as this ensures the object is deleted as soon as it is popped, avoiding the possibility that a leave might occur before it has been deleted, causing a memory leak.

A compound object that owns pointers to other C-class objects should delete those objects in its destructor. Therefore any object which is pointed to by member data of another object (rather than by an automatic variable) does not need to be pushed to the cleanup stack. In fact, it **must not** be pushed to the cleanup stack, or it will be destroyed twice if a leave occurs: once by the destructor and once by the TRAP macro.

2.4 RULE 3: TWO-PHASE CONSTRUCTION

Sometimes, a constructor will need to allocate resources, such as memory. The most ubiquitous case is that of a compound C-class: if a compound class contains a pointer to another C-class, it will need to allocate memory for that class during its own construction.

(Note: C-classes in Symbian OS are always allocated on the heap, and always have `CBase` as their ultimate base class)

In the following example, `CMyCompoundClass` has a data member which is a pointer to a `CMySimpleClass`.

Here's the definition for `CMySimpleClass`:

```
class CMySimpleClass : public CBase
{
public:
           CMySimpleClass();
           ~CMySimpleClass();
           …
private:
           TInt        iSomeData;
};
```

Here's the definition for CMyCompoundClass:

```
class CMyCompoundClass : public CBase
{
public:
           CMyCompoundClass();
           ~CMyCompoundClass();
           …
private:
           CMySimpleClass          *           iSimpleClass;            // owns
another C-class
};
```

Consider what you might be tempted to write as the constructor for `CMyCompoundClass`:

```
CMyCompoundClass::CMyCompoundClass()
{
           iSimpleClass=new CMySimpleClass;                // WRONG
}
```

Now consider what happens when you `new` a `CMyCompoundClass`, e.g.:

```
CMyCompoundClass* myCompoundClass=new (ELeave) CMyCompoundClass;
```

With the above constructor, the following sequence of events occurs:

1. Memory is allocated for the CMyCompoundClass
2. CMyCompoundClass's constructor is called
3. The constructor news a CMySimpleClass and stores a pointer to it in iSimpleClass
4. The constructor completes

But, if stage 3 fails due to insufficient memory, what happens? There is no way to return an error code from the constructor to indicate that the construction was only half complete. `new` will return a pointer to the memory that was allocated for the `CMyCompoundClass`, but it will point to a partially-constructed object.

If we make the constructor leave, we can detect when the object was not fully constructed, as follows:

```
CMyCompoundClass::CMyCompoundClass()                       // WRONG
{
           iSimpleClass=new (ELeave) CMySimpleClass;
}
```

However, this is not a viable method for indicating that an error has occurred, because we have already allocated memory for the CMyCompoundClass. A leave would destroy the pointer (this) to the allocated memory, and there would be no way to free it, resulting in a memory leak.

The solution is to allocate all the memory for the components of the object, **after** the compound object has been initialized by the C++ constructor. By convention, in Symbian OS this is done in a `ConstructL()` function.

Example

```
void CMyCompoundClass::ConstructL()                        // RIGHT
{
iSimpleClass=new (ELeave) CMySimpleClass;
}
```

The C++ constructor should contain only initialization that cannot leave (if any):

```
CMyCompoundClass::CMyCompoundClass()                            // RIGHT
{
           …                                            // Initialization that cannot
leave
}
```

The object is now constructed as follows:

```
CMyCompoundClass* myCompoundClass=new (ELeave) CMyCompoundClass;
CleanupStack::PushL(myCompoundClass);
myCompoundClass->ConstructL();                                  // RIGHT
```

This can be encapsulated in a `NewL()` or `NewLC()` function for convenience.

2.4.1 Implementing two-phase contruction in NewL() and NewLC()

If a compound object has a `NewL()` function (or `NewLC()`), this should contain both stages of construction. After the first stage, the object should be pushed to the cleanup stack before `ConstructL()` is called, in case `ConstructL()` leaves.

Example:

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

2.4.2 Glossary definitions

Term	Meaning
Leave	A call to `User::Leave()`. This causes program execution to return immediately to the trap harness within which the function was executed.
Trap harness	A macro to which control is returned when a leave occurs (if the leave occurred in code executed inside the harness).
Cleanup stack	A stack containing all the objects that need deleting when a leave occurs.

2.5 Common mistakes

2.5.1 Misuse of TRAP and TRAPD

Some classes repeatedly include code of the form

```
void NonLeavingFunction()
{
           TRAPD(error, LeavingFunctionL());
}
```

Whilst this is legitimate code, it should NOT be widely used. A TRAP is expensive in terms of executable binary size, execution speed, and unless used very carefully can lead to errors being lost. Often a TRAP is used where it would simply be better to add an “L” to the end of the function name and allow the Leave to propagate up.

In particular, the following code is particularly bad; the whole TRAP is meaningless!

```
void NonLeavingFunction()
{
           TRAPD(error, LeavingFunctionL());
           If (error!=KErrNone)
                       User::Leave(error);
}
```

2.5.2 Misuse of new

This code is illegal and dangerous;

```
void NonLeavingFunction()
{
bar* foo = NULL;
           TRAPD(error, foo = new bar());
           foo->DoSomething();
}
```

In this situation it is essential to use `new (ELeave)` as `new` on its own will not leave. This could lead to both memory leaks and use of an uninitialised pointer.

2.5.3 Misuse of “L” suffix

```
void NonLeavingFunction()
{
LeavingFunctionL();
bar* foo = new (ELeave) bar();
bar* foo1 = bar::NewL();
}
```

All three lines of this function break the “L” suffix rule. There are 2 options here; either the leaving lines must be TRAPed (probably not the right solution) or the NonLeavingFunction should become an L function (probably better).

2.6 Memory leaks

It is very important to carry out memory testing regularly during the development of Symbian OS code. If a memory leak is discovered, it is much easier to locate it within the context of work that is currently being done, rather than having to search through the whole app.

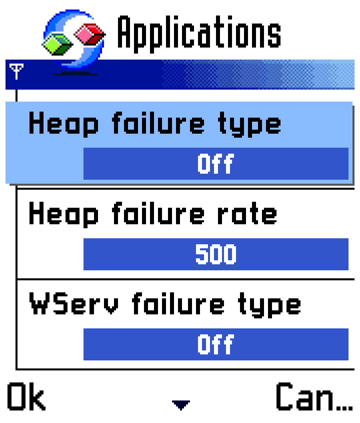
Symbian OS provides heap failure tools in debug builds which can be used to assist in the stress testing of memory in Symbian OS code. In doing this we are looking for 2 aspects of the app’s behaviour;

1. Behaviour of the app when memory runs out
2. Memory leakage, reported when app is closed

The aim should be for the minimum behaviour of user informed with no data loss. Note that it is particularly important to actually close the application using the BACK cba when memory testing. Closing the emulator directly using the top-right close button will not allow the memory checking code to run.

2.6.1 Using tools in the wins emulator

The wins emulator provides a tool for checking memory behaviour on pressing CTRL-SHIFT-ALT-P. It is detailed in the Professional Symbian Programming book, page 158. The utility described in the book is available in Series 60, although it looks slightly different:



Heap failure type
Off
Heap failure rate
500
WServ failure type
Off

2.6.2 Debugging a memory leak

Debugging a memory leak in Symbian OS can be daunting, but there are techniques for doing it that make the process fairly painless. However, memory leaks are never trivially easy to find, and prevention is the best cure! The following tips will prevent leaks appearing in the first place, and ease the pain of searching them out later;

- Understand the CleanupStack & Leave/TRAP paradigm
- Build & run code regularly — if a leak appears, you'll know where it was caused
- Code reviews are very useful
- Use Symbian OS heap checking macros
- When testing, exit the app — don't just kill the emulator

There are 2 types of memory leak. A “Static” leak is one which always occurs repeatably when the application is run – it is caused by a mismatched new & delete. These are relatively easy to find as they always happen in the same place and thus can be debugged. A “Dynamic” leak is one which is non trivial to repeat. For example one caused by an error condition, or race condition. This is more tricky to find, as not all paths through the code cause it.

The wins emulator provides two tools to help solve memory leaks.

2.6.2.1 What Leaked?

When an app is closed, the emulator will panic if memory has leaked (this is actually a _UHEAP_MARKEND macro running). An app should ALWAYS exit clean, even in development. If a panic on close appears whilst you are developing, fix it straightaway. If it is left, it will be 10 times more difficult to track down later.

Under the debugger the panic appears as a Developer Studio dialog: "User Breakpoint called from code at location 0xxxxxx". The stack trace (use View-DebugWindows-CallStack) shows that it's in the destructor of CCoeEnv.

Press OK and F5 to continue. You will get another User Breakpoint, this time at DebugThreadPanic. The output window now shows Panic ALLOC and the address above. Select this address as shown below, and copy it to the clipboard (Edit-Copy).

This is the hexadecimal address of the memory cell that hasn't been deallocated. From this address, you can find out the type of the leaked class by attempting to cast the address to a few possible types. Use the watch window in Visual Studio, and try to cast the pointer badCell to the following types;

- `CBase*` (in case it is a CBase derived object)
- `TDesC16*` (in case it is a string)

In case neither of these casts give any useful info, you have to look elsewhere. It could be an R-Class, although the server should normally panic if a client is not closed. Alternatively it could be a T-Class which is on the heap. Note that this technique could give slightly skewed information if a large compound C-Class has leaked, as it is likely that one of the members of the large class will be reported, rather than the parent class itself.

2.6.2.2 Where was it allocated?

Once the address of the leaked memory is known, you can find where it was allocated by setting a conditional breakpoint in the heap allocator function.

All heap memory allocation goes through the function RHeap::Alloc(int). So start by putting a breakpoint there. Symbian don't currently supply the source code for this function, but you can set the breakpoint explicitly using Edit-Breakpoints-BreakAt:

Use Debug-Go (F5) to continue until the system's first allocation. You won't be able to view the source code, but you can see the disassembled code. Scroll down through the disassembled code, passing the lable 'retryAllocation', until the line before the start of the next function, 'roundToPageSize'. Put a breakpoint on the RET line before it, as shown below:

At that point, register EAX will contain the return value from the RHeap::Alloc function. Use Edit-Breakpoints to set up a breakpoint when it's equal to your offending cell. First disable the breakpoint at RHeap::Alloc, then select the breakpoint you just set and use Condition to set the condition that the return value be the cell you're tracking (note the extra brackets; they seem to be necessary in VC5).

Click OK to both dialogs, then use Debug-Go to continue. Run the application as before; when execution stops at the breakpoint examine the stack to see where the offending cell was allocated.

Note that sometimes the same cell may be allocated and deallocated a few times; it's only the last allocation you're interested in.

If the cell doesn't get allocated, it's may be because this run was a bit different from the first one, and the leaking cell was at a different location. Continue until application exit, and find a new offending cell address. Then set that as an additional Breakpoint at the same location as the other but with Condition to catch the new offending cell, and use Debug-Restart to restart. You may get the error message "Cannot restore all the breakpoints"; this is because the EUSER DLL isn't loaded when EPOC.EXE first starts; the solution is to reenable the RHeap::Alloc(int) breakpoint, run until that is called, and then restore the other, conditional, breakpoints.

Note that code slows down a lot with this breakpoint in place - so enable it at the latest moment! Also, the same address may be allocated many times - which one is the leak? This involves investigating the call stack each time the breakpoint is hit to find the context.

2.7 Asserts and Panics

Many bugs and other problems can be prevented by using ASSERT_DEBUGs. Use them liberally to check for silly parameters coming into functions, null pointers and other error conditions. Many error conditions will not trip up the application straight away, but will lead to other side effect problems later. If errors are caught as soon as they occur, debugging becomes much easier later.

Eg:

```
CMyClass::Function(CThing* aThing)
{
__ASSERT_DEBUG(aThing, Panic(EMyAppNullPointerInFunction));
…
}
```

3. System Resource Usage (ROM and RAM)

3.1 Importance

Phone is a resource limited device. There is a great deal of functionality in the product, and this places great demands on the available system resources. Everyone needs to be aware of these constraints, and attempt to use as little as possible of these limited resources.

3.2 Reducing Code Size

It is important that finished compiled code is as small as possible, to maximise the available space on the device. The following tips will hopefully provide some guidance on how to ensure that no space is being wasted; please take the time to check your code for this kind of thing, and think about other ways in which the compiled size could be reduced.

3.2.1 Unnecessary Exported functions

When functions are exported using IMPORT_C / EXPORT_C from a dll, they use up space for the export table. Only functions which need to be used outside of your dll need to be exported.

3.2.2 Copy and Paste

Copy and Paste often leads to code bloat. When reusing code from other modules, you need to ask the following questions;

- Is this code actually needed?
- Is too much code being copied for the task you need to do?
- Could it be better to abstract the function into a base class or helper module so it can be used from more than one place, rather than copy it?
- Could the code be rewritten more efficiently for the required task, rather than copying something that is nearly what is required?

3.2.3 Obviously Decomposable Functions

There are many places where a number of functions are present in a class, which all do very similar things. Often this code can be abstracted out into a single function which is parameterised to perform the different tasks required. A common example of this kind of thing is a class which implements both NewL and NewLC. Rather than duplicate the code in both functions, NewL can just call NewLC, performing a CleanupStack::Pop afterwards.

3.2.4 Excessive TRAPs

TRAPs use up space when they are compiled. Code which contains many TRAPs (eg more than 5 in a class) is using up too much space. It is also probably designed wrong, as TRAP is not intended to be used extensively in normal code; it is there to allow advanced development to implement special error handling and recovery routines.

3.2.5 Debug code in Release

If there is any code for logging, or debugging or testing, it needs to be excluded in release builds. `#ifdef _DEBUG` can be used for this purpose.

3.2.6 Unneccessary Virtuals

Unnecessary virtual functions are bad, for similar reasons to EXPORTS, as they create extra vtable functions.

3.2.7 Use Common Controls

If at all possible use controls which are available in Avkon (or other shared dll's) instead of developing your own.

3.2.8 Misuse of _L

_L is now deprecated and replaced by the more efficient _LIT, which should be used instead.

3.3 Reducing RAM usage

There are a number of ways to reduce RAM usage. Some methods for doing this (such as bitfields) can obfuscate code, so there is often a compromise to be struck between the reduction in RAM usage and the increase in complexity of the code.

3.3.1 Use bitfields rather than many TBools

In classes which contain large amounts of state or settings data, consider using bitfields to store it. Every TBool requires 32 bits of RAM, but that same 32 bits could hold 32 boolean values in a bitfield. As mentioned above, compare the potential benefits of this with the added code complexity.

3.3.2 Use caution with array granularities

All Symbian OS CArray’s take a granularity. This is to make code more efficient by only allocating space for the array in certain size chunks. This is effective, but some thought needs to go into the choice of granularity. If an array is needed for typically 5 to 8 objects, then a granularity of 4 or 5 would be sensible. If the array always contains 15 objects, then the granularity should be 15. However, if there are typically 2 or 3 objects, having a granularity of 100 would be silly. Similarly, if there are typically 101 to 105 objects, a granularity of 100 would also be silly as it would mean that 200 spaces were always allocated. However a granularity of 1 would also be silly as it would involve many re-allocs. The final choice depends on the usage pattern.

3.3.3 Avoid global data

Don't use global data. Use local variables instead of member variables where they are only used in 1 function.

3.3.4 Beware of base class member data

If writing a base class that it is intended will be widely used, use caution with member data. Do not add member data which is only useful to some derived classes, as every derived class will have no choice but to own it. Take care to only include truly generic member data.

3.3.5 Use the CleanupStack correctly

If the CleanupStack is used correctly, there should be no memory leaks in the code – which in itself guarantees that the application is not using more RAM than it needs to.

3.3.6 Delete early

If temporary objects are allocated on the heap, delete them early, as soon as they are no longer useful. If the life of temporary objects is longer than they are needed then the typical RAM consumption of the application will be higher than it needs to be. Remember though, if a temporary object is deleted whilst a pointer to it is still in scope, the pointer should be set to NULL to prevent and illegal usage or double deletions.

3.3.7 Test on hardware with the maximum data set

If a data set has an upper limit then test on hardware with the maximum data set. It is very easy not to notice that some operation is very slow, or causes problems, if it is never tested on hardware at its limits. Always be sure to load up test scenarios with large data sets.

3.3.8 Break up long complex operations

Displaying long lists on screen can be a source of heavy RAM usage and bad performance while the lists are initialised (eg. a list of all contacts\notes on the device). They can be avoided by writing specific controls which only populate with the entries which are visible on the screen. Scrolling then deletes the entries that leave the screen and adds new ones.

3.4 Reducing Stack usage

The stack available to an application on the target hardware is smaller than the giant stack available to the emulator in the Windows NT environment. This can lead to code which appears to run perfectly in wins not running on the hardware, with apparently random panics. Reducing stack usage is not easy, but there are a couple of things to watch out for

3.4.1 Use descriptors correctly

There are two types of descriptors, heap descriptors (HBufC*) and stack descriptors (TBuf). All descriptors use one of these types for storage. When the stack overflows, 90% of the time it is caused by large descriptors on the stack. Be aware of operations that cause implicit copying of descriptors, and avoid them where possible. It may be better in some situations to allocate HBuf's to work on rather than TBuf's.

Some Symbian OS classes such as TParse can use a lot of stack, and provide alternative versions which use lower stack (TParseBase).

Passing descriptors by reference is preferred to passing by value.

3.4.2 Beware of recursion, and build in limits

If recursive programming is necessary, be aware of the demands on the stack. Try to minimise the size of the parameters being passed down, and try to move local automatic variables out of the recursive part of the function. If at all possible, build in a recursion depth limit to prevent the stack overflowing.

3.4.3 Beware of logging code

Logging code typically involves the formatting of long descriptors and their writing out to files. For this reason they are often the cause of stack overflows.

3.5 Low disk handling

To monitor free space of FFS (Flash File System, alias drive C:) there are two levels defined by the system: Warning Level (WL) and Critical Level (CL). When free disk space meets the WL or CL, the system (EikSRvUI) will display a global note which first warns a user about the situation. Applications & Servers can ignore the WL and concentrate on the CL.

All the operations which fetch or create file on disk with known size N, **must** check the CL first, with N as a parameter to "FFSSpaceBelowCriticalLevelL" method. This check means that disk space is already below critical level or it is going to go below critical level. In case that happens (FFSSpaceBelowCriticalLevelL returns ETrue) applications **must** not write the data, and they need to indicate to a user about the full disk situation. This can be done by leaving with KErrDiskFull error code.

All the operations which fetch or create a file on disk with unknown size must check the CL first with a good estimate or 0 (the default) as parameter to "FFSSpaceBelowCriticalLevelL" method. The zero can be used to check whether we are already below the critical level.

Difficult cases are ones where single items are created in databases, for example contacts. In these cases an estimate can be used of the increase in database the item add would cause.

To support CL check there is a utility method in SysUtil.h of SysUtil.dll. Use that to check the CL. The CL check API looks like the following:

```
/**

* Checks if the free FFS (internal Flash File System) storage
* space is or will fall below Critical Level (CL).
* The CL and FFS drive letter is defined by this module.
* @param aFs File server session.
*    Must be given if available in the caller,
*    e.g. from EIKON environment.
*    If NULL this method will create a temporary session for
*    a check, but then the check is more expensive.
* @param aBytesToWrite number of bytes the caller is about to add
*    FFS, if known by the caller beforehand.
*    The default value 0 checks if the current
*    space is already below the CL.
* @return ETrue if storage space would go below CL after adding
*     aBytes more data, EFalse otherwise.
*     Leaves on error.
*/
IMPORT_C static TBool FFSSpaceBelowCriticalLevelL(
             RFs* aFs, TInt aBytesToWrite = 0);
```

4. Building for ARM Targets

4.1 General Issues

Building for THUMB will in general be more difficult than building for wins and it is normal to find additional compiler errors and warnings from gcc on the first attempt. This is firstly because gcc is in many situations stricter than the Microsoft compiler, and also has some subtle differences which will come out the first time a THUMB build is attempted. The following sections cover a few of the most common pitfalls.

4.2 Function exports

The gcc toolchain is stricter than the wins one when it comes to specifying exported functions. The correct way to export a function from a dll is as follows;

In the header file;

```
Class CMyClass : public CBase
           {
           IMPORT_C void Function();
}
```

And then in the CPP file;

```
EXPORT_C void CMyClass::Function()
           {
           …
}
```

The wins toolchain does not mind if you forget to put the EXPORT_C in the CPP file; it figures out what you mean and exports the function anyway. However, the gcc toolchain requires the IMPORT_C and EXPORT_C to be perfectly matched. If they are not, the function will not be exported from the dll, which will eventually lead to errors such as “Cannot Find Function” when attempting to link other dlls to this one.

4.3 ‘MyDll.DLL has (un)initialised data’ error from PETRAN

The Symbian OS architecture does not allow DLLs to have a data segment (initialised or uninitialised). There are problems with deciding quite what such a data segment would mean (Is it shared by all users of the DLL? Should it be copied for each process which attaches to the DLL?) and significant runtime overheads in implementing any of the possible answers. Symbian OS does provides a mechanism whereby a DLL can manage private storage on a per-thread basis: see the description of the Dll class, in particular Dll::SetTls() and Dll::Tls().

A WINS build uses the underlying Windows DLL mechanisms that can provide per-process DLL data using copy-on-write semantics: this is why the problem goes undetected until the code is built for an actual Symbian OS device.

Consider this section of C++ code, added to a file QSORT.CPP which is part of ESTLIB.DLL.

```
// variables

struct div_t                        uninitialised1;
           // in .DATA
static struct div_t    uninitialised2;                                // in .BSS
struct div_t                        initialised1 = {1,1};             // in
.DATA
static struct div_t    initialised2 = {2,2};              // in .DATA

// constants

const struct div_t                  const1 = {3,3};
const static struct div_t           const2 = {4,4};

const TPoint none(-1,-1);

static const TText* plpPduName[12] =
    {
    _S("Invalid"), _S("DataFlowOff"), _S("DataFlowOn"),
    _S("ConnectRequest"), _S("ConnectResponse"), _S("ChannelClose"),
    _S("Info"), _S("ChannelDisconnect"), _S("End"), _S("Delta"),
    _S("EndOfWrite"), _S("PartialWrite")
    };
```

When this code is built, the messages from PETRAN look something like:

```
PETRAN - PE file preprocessor V01.00 (Build 170)
WARNING: Dll 'ESTLIB[10003B0B].DLL' has initialised data.
WARNING: Dll 'ESTLIB[100002C3].DLL' has uninitialised data.
```

The associated .MAP file contains information which helps to track down the source file involved.

Look in Symbian OS\release\arm4\urel\dllname.map

Search for "`.data`" or "`.bss`"

In this example, we find:

```
.data      0x10017000   0x200
        0x10017000        __data_start__=.
 *(.data)
 .data      0x10017000    0x40 ..\..\Symbian
OS\BUILD\STDLIB\BMMP\ESTLIB\ARM4\UREL\ESTLIB.in(QSORT.o)
        0x10017000        initialised1
 *(.data2)
 *(SORT(.data$*))
        0x10017040        __data_end__=.
 *(.data_cygwin_nocopy)

.bss      0x10018000    0x18
        0x10018000        __bss_start__=.
 *(.bss)
 .bss      0x10018000    0x18 ..\..\Symbian
OS\BUILD\STDLIB\BMMP\ESTLIB\ARM4\UREL\ESTLIB.in(QSORT.o)
        0x10018008        uninitialised1
 *(COMMON)
        0x10018018        __bss_end__=.
```

So the DLL has 0x18 bytes of uninitialised data (.BSS) and 0x40 bytes of initialised data (.DATA), all of which comes from QSORT.o

The variables "initialised1" and "uninitialised1" both have global scope, so the MAP file lists them by name (and puts them both in the initialised data). The static variables don't get named in the .MAP file.

Removing the first four lines of code shown above leaves just variables which are declared as "const", but only reduces the .BSS to 0x08 bytes and the .DATA to 0x30 bytes. There are two problems remaining:

Declaring C++ objects as "const" doesn't help if they have a constructor. The 8 bytes of uninitialised data are allocated to hold the TPoint object, but it won't become "const" until after the constructor has been run.

The declaration "`const TText*`" says that the TText values may not be altered, but it doesn't make the pointer a constant as well. The 48 bytes of initialised data are the 12 pointers in the plpPduName array. To make the pointers constant as well as the values they point to, the declaration needs and additional "`const`" after the "`TText*`".

```
static const TText* const plpPduName[12] =
    {
    _S("Invalid"), _S("DataFlowOff"), _S("DataFlowOn"),
    _S("ConnectRequest"), _S("ConnectResponse"), _S("ChannelClose"),
    _S("Info"), _S("ChannelDisconnect"), _S("End"), _S("Delta"),
    _S("EndOfWrite"), _S("PartialWrite")
    };
```

Removing the TPoint global variable and adding the extra "const" to the plpPduName array finally removes the last of the offending .BSS and .DATA.