



Following are the general tips for avoiding unnecessary and common errors, and make your program efficient.

## Contents

- [1 KERN-EXEC 3](#)
- [2 Some common errors for application panic](#)
- [3 Use CleanupClosePushL\(\)](#)
- [4 HBufC](#)
- [5 L\(\) Macro](#)
- [6 TRAP](#)
- [7 Cleanup Stack](#)
- [8 Don't push objects on the cleanup stack twice](#)
- [9 Two-phase construction](#)
- [10 Descriptors as function parameters](#)
- [11 When using Active Objects, be careful of the following things](#)
- [12 Ensure your application responds to system shutdown events](#)
- [13 Make use of the Active Object framework wherever possible](#)
- [14 Program compiles for WINS but not for ARMI](#)
- [15 HTTP Posts](#)

## KERN-EXEC 3

Kern-Exec 3 crashes are often caused due to stack corruption or stack Overflow, prefer the use of heap to the stack. Be aware that recursive functions can eat the stack at runtime ? this will lead to a Kern-Exec 3 panic.

## Some common errors for application panic

? Failure to have properly added a non-member, heap-allocated variable to the Cleanup Stack. ? The ?double delete? ? e.g. failure to correctly Pop() an already destroyed item from the Cleanup Stack, causing the stack to try to delete it again at a later time. ? Accessing functions in variables which may not exist in your destructor;

e.g.

```
CSomeClass::~~CSomeClass()  
{  
    iServer->Close();  
    delete iServer;  
}
```

The above statement should always be coded as

## How\_to\_avoid\_common\_errors\_and\_make\_program\_efficient.

```
CSomeClass::~CSomeClass ()
{
if (iServer)
{
    iServer();
    delete iServer;
}
}
```

? Putting member variables on the Cleanup Stack ? never do this, just delete them in your destructor as normal.

## Use CleanupClosePushL()

Always use CleanupClosePushL() with R classes which have a Close() method. This will ensure they are properly cleaned up if a leave occurs.

For example:

```
RFile file;
User::LeaveIfError(file.Open(.....));
CleanupClosePushL(file);
// some code
CleanupStack::PopAndDestroy();
```

## HBufC

Always set member HBufC variables to NULL after deleting them. Since HBufC allocation (or reallocation) can potentially leave, you could find yourself in a situation where your destructor attempts to delete an HBufC which no longer exists.

You don't need to use HBufC::Des() to get into an HBufC. All you have to do is dereference the HBufC with the \* operator ? this is particularly relevant, for example, when passing an HBufC as an argument to a method which takes a TDesC& parameter.

## \_L() Macro

Dont use the \_L() macro in your code. you should prefer \_LIT() instead. The problem with \_L() is that it calls the TPtrC(const TText\*) constructor, which has to call a strlen() function to work out the length of the string. While this doesn't cost extra RAM, it does cost CPU cycles at runtime. By contrast, the \_LIT() macro directly constructs a structure which is fully initialized at compile time, so it saves the CPU overhead of constructing

## How\_to\_avoid\_common\_errors\_and\_make\_program\_efficient.

the TPtrC.

Alternatively, you can use following macros instead of \_L():

```
#define __L(a) (TPtrC((const TText *)L ## a, sizeof(L##a)/2-1))
#define __L8(a) (TPtrC8((const TText8 *) (a), sizeof(a)-1))
#define __L16(a) (TPtrC16((const TText16 *)L ## a, sizeof(L##a)/2-1))
```

It gets rid of strlen, but still TPtrC constructors are not inline functions.

## TRAP

If you have cause to use a TRAP of your own, do not ignore all errors. A common coding mistake is:

```
TPAPD(err, SomeFunctionL());
if (err == KErrNone || err == KErrNotFound)
{
    // Do something else
}
```

This means all other error codes are ignored. If you must have a pattern like the above, leave for other errors:

```
TPAPD(err, SomeFunctionL());
if (err == KErrNone || err == KErrNotFound)
{
    // Do something else
}
else
    User::Leave(err);
```

## Cleanup Stack

Do not wait to PushL() things on to the Cleanup Stack. Any newly allocated object (except member variables) should be added to the stack immediately. For example, the following is wrong:

```
void doExampleL()
{
    CSomeObject* Object1=new (ELeave) CSomeObject;
    CSomeObject* Object2[[Category:Symbian C++]]2=new (ELeave) CSomeObject;
}
```

because the allocation of Object2 could fail, leaving Object1 ?dangling? with no method of cleanup. The above should be:

```
void doExampleL()
{
    CSomeObject* Object1=new (ELeave) CSomeObject;
    CleanupStack::PushL(Object1);
    CSomeObject* Object2=new (ELeave) CSomeObject;
    CleanupStack::PushL(Object2);
}
```

## Don't push objects on the cleanup stack twice

Always remember that functions with a trailing C on their name automatically put the object on the Cleanup Stack. You should not push these objects onto the stack yourself, or they will be present twice. The trailing C functions are useful when you are allocating non-member variables.

## Two-phase construction

Two-phase construction is specially designed to avoid memory leaks, It is essential that you implement this design pattern to avoid memory leaks in your code. For each line of code you write, a good question to ask yourself is "Can this line leave?". If the answer is "Yes?", then think: "Will all resources be freed?".

## Descriptors as function parameters

When using descriptors as function parameters, use the base class by default. In most cases, pass descriptors around as a const TDesC&. For modifiable descriptors use TDes&.

## When using Active Objects, be careful of the following things

- ? There is no need to call TRAP() inside RunL(). The Active Scheduler itself already TRAPs RunL() and calls CActive::RunError() after a leave.
- ? To this end, you should implement your own RunError() function to handle leaves from RunL().
- ? Keep RunL() operations short and quick. A long-running operation will block other AOs from running.
- ? Always implement a DoCancel() function and always call Cancel() in the AO destructor.

## Ensure your application responds to system shutdown events

It is vital that you respond to EEikCmdExit (and any platform-specific events, for example EAknSoftkeyBack and EAknCmdExit on Series 60) in your AppUi::HandleCommandL() method.

## **Make use of the Active Object framework wherever possible**

Tight polling in a loop is highly inappropriate on a battery powered device and can lead to significant power drain.

## **Program compiles for WINS but not for ARMI**

Your program compiles for WINS even runs on the emulator but gives errors during ARMI build. Possible reason for this is you have left a space in the header file name i.e. instead of `#include "headerfile.h"` you've typed `#include "headerfile.h "`. Just remove that space and the compile again.

## **HTTP Posts**

In case of HTTP posts with forms always remember to delete the instance of `CHTTPFormEncoder`. If it is a php script on your server the Form elements are read bottom to top whereas python script reads the Form elements top to bottom. So what may seem to work on php wont work if the scripting language is changed to python. So always:

```
delete iFormEncoder;  
iFormEncoder = NULL;  
iFormEncoder = CHTTPFormEncoder::NewL();
```