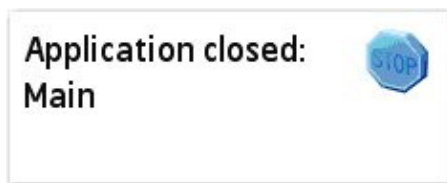


Contents

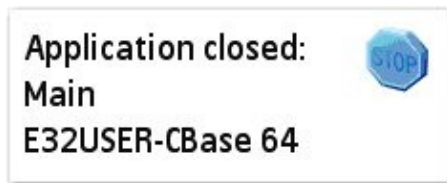
- [1 The emulator doesn't display the details of a panic](#)
- [2 Bug detection using assertions](#)
- [3 Detecting memory leaks with `_UHEAP_MARK` and `_UHEAP_MARKEND` macros](#)
- [4 Object invariance macros](#)
- [5 Detecting incorrect use of cleanup stack with expected items](#)

The emulator doesn't display the details of a panic

If a panic occurs, the emulator doesn't display its details unless there is a file named `?ErrRd?` in a specific location. This makes it difficult to know what caused the panic.



Before the 3rd edition of the SDK, the `ErrRd` file must be created manually, but from the 3rd edition onwards it can be found by default from directory `?C:\Symbian\9.2\S60_3rd_FP1\Epoc32\winscw\c\resource?`. With `ErrRd`, the output looks like this, in case a panic occurs:



Tip: If the `ErrRd` file cannot be found even if you are using 3rd edition SDK, start the emulator, select `Tools > Preferences` and check `Extended panic code file`.

Bug detection using assertions

Assertions are used to check that assumptions made about code are correct and that the state, for example, of objects, function parameters or return values, is as expected. There are two assertion macros defined on Symbian OS, `__ASSERT_ALWAYS` and `__ASSERT_DEBUG`. The difference between them is that `__ASSERT_DEBUG` will not affect production code whereas `__ASSERT_ALWAYS` will. Here's an example of how to use the `__ASSERT_DEBUG` macro:

```
void TestValue(TInt aValue)
{
    _LIT(KPanicCategory, "TestValue");
    __ASSERT_DEBUG((aValue >= 0), User::Panic(KPanicCategory, 99));
    // Do something with aValue
    // ...
}
```

```
}
```

In the example above, a panic -99 is raised if the parameter aValue is less than 0.

Note: The assertion macros do not panic by default, allowing you to decide what procedure to call in case assertion fails. Despite this, you should always raise a panic in this case rather than return an error or leave.

Because the example above uses the `__ASSERT_DEBUG` macro, aValue is only tested in debug builds. If it is necessary to test the parameter also in production code, `__ASSERT_ALWAYS` should be used.

When you don't expect external callers to need to trace a panic, an alternative to using `__ASSERT_DEBUG` is `ASSERT` macro. `ASSERT` is exactly like debug assertion macro except that it doesn't need you to provide a panic category or descriptor. Here's the definition of the macro from `e32def.h` file:

```
#define ASSERT(x) __ASSERT_DEBUG(x, User::Invariant())
```

Here's an example of how to use the `ASSERT` macro:

```
void TestValue(TInt aValue)
{
    ASSERT(aValue >= 0);
    // Do something with aValue
    // ...
}
```

Detecting memory leaks with `__UHEAP_MARK` and `__UHEAP_MARKEND` macros

One possibility to check that your code is managing heap memory correctly, in other words not leaking memory, is to use `__UHEAP_MARK` and `__UHEAP_MARKEND` macros. Here is an example:

```
GLDEF_C TInt E32Main()
{
    // Start checking memory leaks
    __UHEAP_MARK;

    // Create a fixed-length, flat array, which contains 10 integers
    CArrayFixFlat<TInt>* fixFlatArray;
    fixFlatArray = new(ELeave) CArrayFixFlat<TInt>(10);
    // Array is not deleted, so memory will leak

    // Stop checking memory leaks and cause a panic if there is a leak
    __UHEAP_MARKEND;

    return KErrNone;
}
```

Due to the array not being deleted and because of memory leak detecting macros, the code example above will cause a panic when the application is closed, as shown in figure below:



It is worth mentioning that heap-checking macros are only compiled into debug builds, so they can be safely left in the production code without having any impact on the code size or speed.

Object invariance macros

There are two macros that allow you to check the state of an object, `__DECLARE_TEST` and `__TEST_INVARIANT`. In practice they are used so that the programmer first creates an invariance test function, which is then called, typically at the beginning and end of a function in which object's state needs to be checked. Here's an example of a class that represents a living person and uses invariance testing to verify that the person has a gender and his or her age is not negative:

```
class CLivingPerson : public CBase
{
public:
    enum TGender {EMale, EFemale};
public:
    CLivingPerson(TGender aGender);
    ~CLivingPerson();
public:
    void SetAge(const TInt aAge);
private:
    TGender iGender;
    TInt iAgeInYears;
    __DECLARE_TEST; // Object invariance testing
};

CLivingPerson::CLivingPerson(TGender aGender) : iGender(aGender) {}
CLivingPerson::~~CLivingPerson() {}

void CLivingPerson::SetAge(const TInt aAge)
{
    // Set age and check object invariance
    __TEST_INVARIANT;
    iAgeInYears = aAge;
    __TEST_INVARIANT;
}

void CLivingPerson::__DbgTestInvariant() const
{
#ifdef _DEBUG // Built into debug code only
    // Person should be either male or female
    ASSERT((iGender == EMale) || (iGender == EFemale));

    // Person's age shouldn't be negative
    ASSERT(iAgeInYears >= 0);
#endif
}
```

}

Due to the use of ASSERT macros in the example above, a USER 0 panic is raised if the object's state is incorrect.

Detecting incorrect use of cleanup stack with expected items

Objects on the cleanup stack should be popped when there is no longer a chance that they would be orphaned by a leave. Thus, popping usually happens just before the objects are deleted. PopAndDestroy function is normally used instead of Pop, as this ensures the object is deleted as soon as it is popped, which avoids the possibility for a memory leak. Both CleanupStack::Pop and CleanupStack::PopAndDestroy have an overloaded form that allows the caller to declare an "expected item", an item that should be popped from the stack. In case expected item doesn't match the popped item, an E32USER-CBase 90 panic is raised. These two overloaded forms are the recommended ones, because they help detect incorrect use of the cleanup stack.

```
CClass* obj = new(ELeave) CClass;
CleanupStack::PushL(obj);
// ...
CleanupStack::PopAndDestroy(obj); // Panics if ?obj? not on top
```

Note: In release builds, the expected item check will be disabled, so release builds are not affected either in binary size or in efficiency from using it.