Testing a Series 60 Application

Version 1.0
August 2002

SERIES 60 PLATFORM

Table of Contents

Glossary Definitions

Term	Meaning
MSVC	Microsoft Visual C++™ the development environment used for Series 60 C++ development
DLL	Dynamic Link Library
SDK	Software Developer Kit
Cleanup Stack	Symbian OS mechanism for ensuring that failed memory allocations do not cause memory leaks
Leave	Symbian OS mechanism for exception handling
API	Application Programming Interface

Legal Notices

1. About This Document

1.1 Purpose

The following document is intended to give an overview of testing methodologies available to the Series 60 developer.

Combining general software engineering mechanisms with Series 60-specific practices, this document is not meant to be exhaustive, but rather, serve as a point of reference to other, more in-depth sources.

1.2 Audience

There is a bias towards the Series 60 C++ developer due to the nature of the provided debugging tools. However the general software engineering principles can be applied to development in any language.

2. Introduction

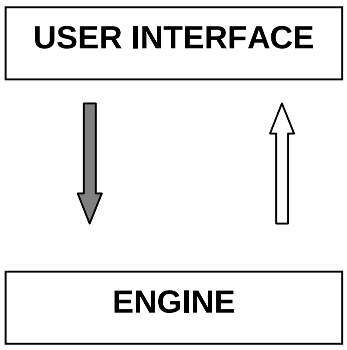
Smartphones are characterized by their limited hardware in terms of memory, screen size and input methods. Consequently, software for a smartphone platform such as Series 60 must be developed with these constraints in mind. This, in turn, will influence testing aims and testing methodologies. Testing is one of the most vital parts of application development and must be performed at all stages of the software development life cycle. Before release to market, applications should undergo a thorough and rigorous testing program. From a general perspective testing should be performed with the initial software requirements in mind and be well planned. Due to the modular nature of development, particularly from the object-oriented perspective so prevalent on Series 60 Platform, testing should be performed on the individual units that comprise an application and then be migrated to the application as a whole.

3. Tools for Development

The development of Series 60 applications is undertaken utilizing the Series 60 emulator and a development environment. For C++ applications, Microsoft Visual C++™ (MSVC) is currently the preferred environment. Java™ applications can be created using any environment capable of generating MIDlets, for example Sun One Studio™ or Borland JBuilder™. Applications are built within the development environment of choice and can then be run and tested within the emulator. The emulator is a representation of the target platform and ensures that the majority of development takes place on a PC. However, it is advisable to build and test applications for hardware as early and frequently as possible, since differences can emerge. When developing for C++, building for the emulator is performed using the Microsoft C++ compiler, while the GCC compiler carries out target building.

4. The Engine/User Interface Architecture

Developers are advised to separate the application engine from the application user interface when developing for Series 60. This serves to enhance portability between Symbian OS-based devices and aids testing. The application engine is seen to contain the data pertaining to the application and is likely to be implemented as a DLL with a public interface so that it can potentially be used by other applications. A two-way relationship exists between the user interface and engine. The user interface receives input that can influence the state of data in the engine, while the user interface is drawn with respect to the data in the engine.



Series 60 application development involves the bringing together of various different aspects of software creation. Development teams can focus on a diverse set of tasks ranging from creating user interface components to creating application engines, all of which will need to interact seamlessly. Consequently, testing techniques emerge that are relevant for the engine, the user interface, or both.

5. Application-Wide Testing Techniques

Regardless of the task at hand, certain testing facilities are applicable to all Series 60 C++ applications and can range from simple, common-sense tasks that are often overlooked, to more complex debugging routines and tools.

5.1 Following Coding Standards and Idioms

The naming conventions outlined within the Series 60 development documentation are extremely useful for discovering the nature and behavior of variables and functions. For example, seeing the prefix “i” in a variable name indicates that this variable is class member data and should not be placed on the Cleanup Stack. While the appearance of a trailing L at the end of a function implies that there is the potential for the function to “Leave”. A developer is advised to create code that adheres to the coding conventions as this will aid interaction with other developers' code and enhance readability, particularly during code reviews and other such coding inspections. Following such conventions can also reduce debugging time since a developer can instantly see how particular variables and functions should be behaving if a defect has been traced to them.

See also Series 60 SDK Documentation – Search: "Coding Conventions"

5.2 Implementing the Correct Exiting Mechanism

A Series 60 application can be exited in one of the following ways:-

- Using the right soft key
- Using the Options menu
- From the application task list

This is handled within a **`HandleCommandL()`** function, by using either the **`EEikCmdExit`** or **`EAknCmdExit`** flag, depending on the context. When exiting via the softkey **`EEikCmdExit`** is used; however, when the Options menu is employed, **`EAknCmdExit`** should be used. The reason for this is because application embedding necessitates two types of exit. On the one hand, there is the need to exit and return control to a parent application (**`EEikCmdExit`**), on the other hand, there is the need to exit parent applications as well (**`EAknCmdExit`**). Developers should note that the right soft key is often also assigned the role of a "Back" key press through the **`EAknSoftkeyBack`** flag. A context management mechanism must be implemented to distinguish between the use of the right soft key for the purpose of exiting or performing a "Back" operation.

See also Series 60 SDK Documentation – Series 60 Application Framework Handbook: "Application Exit and EEikCmdExit"

5.3 Handling Limited Amounts of Memory for Write Operations

Smartphones have a relatively small amount of memory. Consequently, users may often be faced with a situation wherein memory is extremely limited. If an application is performing a write operation it is advisable to check the current state of memory. A function called **`FFSSpaceBelowCriticalLevel()`** should be used to check that there is memory available for the write operation. Testing will require a simulation of almost-full disk space and monitor the behavior of the application when it tries to write to the disk.

See also Series 60 SDK Documentation – Coding Idioms for Symbian Developers: "Low Disk Handling"

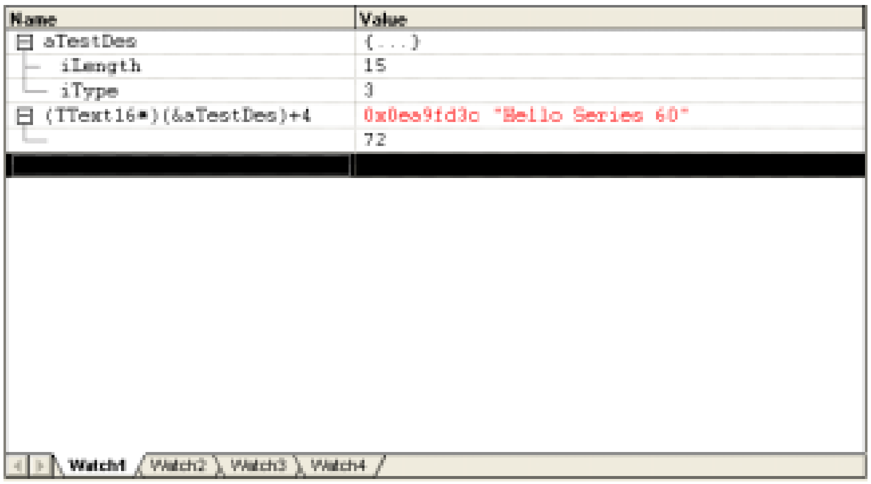
5.4 Checking the Contents of Descriptors

Handling strings and characters when developing for Series 60 is achieved through the descriptor classes. The most useful facility while testing is to be able to view the contents of a descriptor, which can be achieved through the following:

- Insert a breakpoint in the code.
- Ensure that *"Display Unicode Strings"* is checked in the *Tools>Options>Debug* dialog.
- Enter the descriptor variable name into the watch window.
- Expand the value in the watch window.
- Apply one of the following casts to the descriptor name (**DesName**) depending on the value of the iType attribute:

Character Set	**Value Of iType**	**Cast**
Unicode	0	(TText16*)(&**DesName**)+2
	1	(TPtrC16*)&**DesName**
	2	(TPtr16*)&**DesName**
	3	(TText16*)(&**DesName**)+4
	4	(TText16*)(*((int*)&**DesName**+2))+2
Narrow (descriptor type ends in "8")	0	(char*)(&**DesName**)+4
	1	(TPtrC8*)&**DesName**
	2	(TPtr8*)&**DesName**
	3	(char*)(&**DesName**)+8
	4	(char*)(*((int*)&**DesName**+2))+4

The illustration below is of a Unicode TDesC, with an iType value of 3:

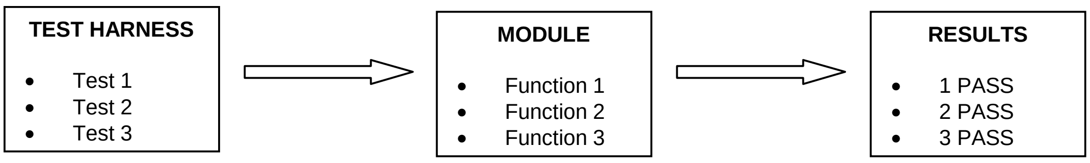


```
Name                          Value
⊟ aTestDes                    {...}
   iLength                    15
   iType                      3
⊟ (TText16*)(&aTestDes)+4     0x0ea9fd3c "Hello Series 60"
                              72
```

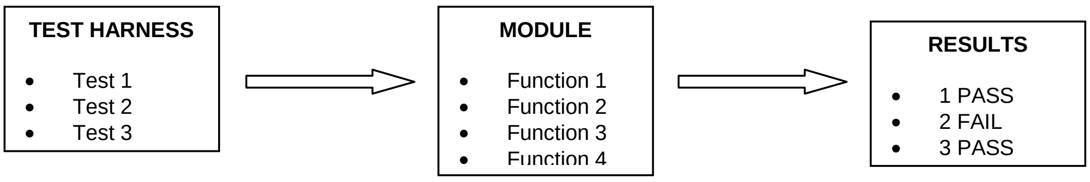
5.5 Regression Testing

As an application develops in complexity throughout its lifecycle, so will the variety and depth of required testing routines. Regression testing is a method for ensuring that additional modules and module enhancements have not corrupted previously verified code. By utilizing tests designed for older functionality new functionality can be tested.

Testing Step 1:



Testing Step 2:



In the above illustration it becomes apparent that the creation of Function 4 introduces a defect in Function 2, and this has been uncovered by previously successful tests. Consequently it is clear how regression testing is useful in testing the effects of new functionality introduced into an application.

5.6 Debug Output

The Series 60 development environment provides a useful function for printing simple debugging information to the debug window of MSVC. This ability facilitates the discovery of the current execution point in the application code and the current values of particular variables. The outputting of information is achieved through the following:

```
RDebug::Print();
```

or in practical terms:

```
RDebug::Print(_L(“Hello I am in the Hello World function!”));
```

This can also be used in conjunction with `printf`-style output formatting. For example, in a spreadsheet application the developer wants to continually keep track of which cell is currently being highlighted for use:

```
TInt currentXCell = iSheet->GetHighlightedXCell();
TInt currentYCell = iSheet->GetHighlightedYCell();

RDebug::Print(_L("Currently cell (%d, %d) is being highlighted.\n"),
currentXCell, currentYCell);
```

5.7 Specialist Macros

Within the Series 60 development tool chain are a series of macros each performing a specific role within the testing process:

- Assertions - Parameter scrutiny
- Heap Operations - Memory checking
- Test Invariant - Object state verification

5.7.1 Assertions

There are two types of Assert macro for checking erroneous data input. `ASSERT_DEBUG` handles errors that are a consequence of development mistakes and the assertion is made only when running the emulator in debug mode. `ASSERT_ALWAYS` is used for exceptions at run-time when the application is in use.

For illustration purposes imagine a situation where a class of mathematical functions performs a division operation where the denominator must be greater than zero.

```
void CTestMath::SetDenominator(TInt aDenominator)
```

```
  {
  _LIT(KZeroOrLessDenominatorPanic, “The denominator is not
  greater than zero”);
  ASSERT_DEBUG(aDenominator > 0,
  User::Panic(KZeroOrLessDenominatorPanic));
      iDenominator = aDenominator;
  }
```

If an **ASSERT_ALWAYS** is required, the above **ASSERT_DEBUG** is simply replaced with:

```
ASSERT_ALWAYS(aDenominator > 0,
User::Panic(KZeroOrLessDenominatorPanic));
```

See also Series 60 SDK Documentation – Search: “Coding Conventions”

5.7.2 Heap Operations

There are numerous macros defined in Series 60 that assist the developer with heap inspection and are particularly useful for out of memory (OOM) testing, i.e., testing the application to see how it and the device respond to memory exhaustion. These are listed below:

Macro Name	Description
`__*HEAP_CHECK(aParam)`	The number of cells allocated at this nested level of the heap is checked against the value of aParam.
`__*HEAP_CHECKALL(aParam)`	Heap should be checked to have allocated aParam heap cells.
`__*HEAP_FAILNEXT(aParam)`	After aParam attempts to perform a heap allocation, a failure takes place.
`__*HEAP_MARK`	Commencement of heap checking
`__*HEAP_MARKEND`	Termination of heap checking
`__*HEAP_MARKENDC(aParam)`	Checking heap ended with a number of heap cells equal to aParam expected to still be allocated.
`__*HEAP_RESET`	Imitation of heap allocation failure is ended
`__*HEAP_SETFAIL(aNature, aFrequency)`	Heap allocation failure imitated. aNature: the nature of the failure that is imitated; aFrequency: the frequency of failure

Where “*” in the table above should be replaced by one of the following letters, corresponding to the different types of heap:

- **`K – Kernel heap, e.g., __KHEAP_MARKEND`**
- **`R – Developer specified heap, e.g., __RHEAP_MARKEND`**
- **`U – Current thread’s heap, e.g., __UHEAP_MARKEND`**

A comprehensive illustration of each type is beyond the scope of this document. However, additional information can be found in the SDK.

See also Series 60 SDK Documentation – Search: “Memory Allocation”

5.7.3 Test Invariant

The methods called on an object have the potential to leave the object in an invalid or unsafe state. To ensure an object is in a safe state the **__DECLARE_TEST** and **__TEST_INVARIANT** macros can be used, with a partnering **__DbgTestInvariant()** function.

For illustration, imagine a class with some member data, which can never be zero. The developer’s first step is to employ the **__DECLARE_TEST** macro by adding it as the last line in the class definition:

```
class TTestingClass
```

```
{
public:
/*public functions etc*/
private:
/*private functions etc*/
TInt iCanNeverBeZero;
__DECLARE_TEST
};

The next step is to create the invariance checking function
in this case:
TTestingClass:: __DbgTestInvariant()
{
#if defined(_DEBUG)
if(iCanNeverBeZero == 0)
User::Invariant();
#endif
}

Then in a particular function, checking can take place:
void TTestingClass::PerformATask()
{
__TEST_INVARIANT
/*Perform task involving some change to iCanNeverBeZero*/
__TEST_INVARIANT
}
```

The first macro checks that **`iCanNeverBeZero`** is in a valid state before proceeding to the function's statements. Once finished, the second macro checks that the statements have not in anyway invalidated **`iCanNeverBeZero`**.

See also Series 60 SDK Documentation – Search: "System Macros".

6. Application Engine Testing Techniques

6.1 Use of Console Applications for Testing Engines

A common Series 60 development idiom is the separation of the application user interface from the application engine. Series 60 development tools provide a console environment, which is ideal for displaying the output of engine calculations without the need to negotiate the complexities of developing a user interface. This in itself could introduce more defects while distracting the developer from the matter at hand – testing the application engine.

To uncover defects, the developer should concentrate on developing test harnesses that automate the testing process and, if possible, output results to both the console and external files for inspection and comparison. If the functions of an application have been verified through a simple console test harness, then debugging time is further reduced when a user interface is introduced and defects arise. Having had the test harness in place it is more than likely that the defects are on the user interface side and the time it takes to locate the defect is reduced.

7. Application User Interface Testing Techniques

7.1 User Interface Component Creation and Testing

Developers can create their own user interface components – for example, a text editor with an automatic spell checker – to augment those provided by Series 60 and to perform a specific function. Testing of these components should focus on behavior and display. Concerns will center on resizing consistency, interaction with other controls, event management, and input handling. At the same time, construction of such controls should be possible through resource files or API, but this will also require scrutiny. These issues are very important if the control is to be used by other developers or even made publicly available.

7.2 Debug Keys

The Series 60 emulator provides the developer with a series of key combinations in debug mode that can greatly assist in testing user-interface-driven applications. These can perform tasks that range from resource allocation to screen refreshing.

For example, using **Ctrl** + **Alt** + **Shift** +:

- **T** - shows currently active programs in a task list.
- **R** - redraws the entire screen in order to test that applications can cope with redraws.
- **A** - indicates the amount of cells the application has currently allocated on the heap
- **B** - illustrates the quantity of file server resources currently utilized by the application.

A more comprehensive list can be found in the SDK

See Series 60 SDK Documentation – Search: “Debug Facilities“.

7.3 Internationalization

Series 60 smartphones will be available throughout the world, and many users will have language settings to suit their geographical location. The requirements for internationalization need to be a design consideration for developers of Series 60 applications. A simple problem on the user-interface side is that words in different languages may have a different number of letters and controls will have to be resized to accommodate this; however, it may result in the application redrawing incorrectly. For example, the word “Close” in English is “Schliessen” in German.

8. Testing on Hardware

In time, a variety of Series 60 devices will emerge, each with unique features.

Despite this variety, certain general considerations are applicable to all device profiles:

- Differences in user interaction between the emulator and target hardware. For example use of the PC keyboard to input data is not the same as data input using a smartphone keypad
- Scope for “hardware dependent” behavior differences between the emulator environment and a physical device: For example, the granularity of clock ticks is not the same, so applications that rely on certain timing operations may have to make compensations
- Call handling and other such situations that are not immediately evident during emulator-based development. For example, analyzing how an application behaves when the smartphone receives a telephone call
- A call stack size limit of 8k, which can be exceeded by an application running in the emulator but not by an application running on the target hardware
- Target hardware provides a multi-process environment, whereas the emulator provides a single process
- Certain actions are possible in the emulator but not on target hardware: For example, it is possible to display an information message using

`CCoeEnv::InfoMsg()` within the emulator environment, but it will not be displayed on the target hardware

All applications should be tested on hardware as early as possible to alleviate such problems, or at the very least to provide experience in handling some of the more complex situations.

9. General Testing Perspectives

Developers should give due consideration to testing issues that are of a general nature, which can greatly assist development on Series 60 Platform. These issues fall into the following categories:-

- Stress testing
- Recovery testing
- Volume testing
- Asynchronous testing

9.1 Stress Testing

Stress testing determines how an application behaves in situations where system resources are limited. A simple example is to run several applications on the device, at the same time as the tested application. This can often result in unanticipated situations, that need to be evaluated carefully and potentially handled. Out Of Memory testing (see **Section 8.6.2 Heap Operations**) is also a form of stress testing.

9.2 Recovery Testing

Recovery tests ascertain what will happen to an application following unexpected situations. Examples include terminating a connection while the application is retrieving information from a remote source or removing the battery when the application is processing information. The expected results of such tests would be that the application reconnects, proceeds from the last known saved instance, or restarts completely.

9.3 Volume Testing

Volume testing assesses the application behavior when a certain action is repeated numerous times, e.g. the response of the application if ten short messages are received in a row or it is repeatedly opened and closed.

9.4 Asynchronous Testing

Asynchronous tests expose the application to a variety of system-oriented functions. These functions can include the following:-

- Incoming communication, e.g., receiving voice calls and short messages
- Outgoing communication, e.g., initiating a voice call and sending information via infrared
- System alarms, e.g., clock alarms and calendar alarms
- System notifications, e.g., battery full

An application should gracefully handle all of the functionality that is specific to the target device. To illustrate, a game must pause itself and save its current state if a telephone call is received.