TRUSTONIC

<t-base Integration Manual

<t-base

PREFACE

VERSION HISTORY

Version	Date	Status	Modification
1.0	05 January 2013	Issue	First version
1.1	22 May 2013	Issue	Update for <t-base-202
1.2	10 Oct 2013	Issue	Minors fixes (and ready for t-base-3xx)

TABLE OF CONTENTS

1 INTRODUCTION

This document explains how to integrate <t-base on a platform. <t-base integration consists in:

- Integrating the <t-base Normal World components for Android
- Integrating the <t-base Secure World components
- Running the Validation Test Suite to verify the integration.

Note that in order to enable <t-base-2xx and the OTA-Operated Containers on a Device, the device manufacturer must install and integrate Trustonic's Key Provisioning Host (KPH) at its manufacturing line. The KPH is a tool which injects the TEE Authentication Token on the device and which stores a copy in the Trustonic backend system. This process and the integration steps are specified in a separate document (*t-base_Provisioning_Manual.pdf*).

The <t-base product package is structured as follows:

- *Documentations*, this directory regroups the integrators documentations (how to integrate <t-base into a device, how to use <t-base provisioning tools).
- *AndroidIntegration*, the directory which contains all the Normal World components.
- *SecureIntegration*, the directory which contains all the Secure World components (<t-base core binary and Trustonic System Trusted Applications).
- *t-base-dev-kit*, the <t-base development kit, with documentations for Trusted Applications developers and samples.
- *ValidationTestSuite*, a basic test suite which will allow integrators to quickly validate that all the critical features of <t-base have been successfully integrated.

2 <t-base Normal World Components

Normal World components of <t-base are:

In Android User space:

- The <t-base Daemon: component in charge of registry and initialization of Normal World/Secure World communication.
- The Root Provisioning Agent: component that is used when installing service provider and developer trusted applications (it includes the Curl library, for XML manipulations and a log library).
- The <t-base Client library: user space APIs available for Normal World application developers.
- The <t-base registry interface: management of the registry.
- The Provisioning library: library used for keys provisioning at the device manufactory.

In Linux Kernel space:

- The <t-base Linux driver: component in charge of communication between the Normal World user space client and the Secure World (with two interfaces available: one reserved for the Daemon and one allocated to direct communication with all the user space clients).
- The <t-base Kernel API: kernel level APIs available for Normal World driver developers.

All the Normal World components of <t-base product are provided in source code and in binary. They are all grouped in directory *AndroidIntegration*.

The purpose of the *AndroidIntegration* folder is to provide a simple way of integrating the Normal World components of <t-base in an Android source tree.

2.1 BUILDING NORMAL WORLD COMPONENTS

The *AndroidIntegration* folder contains 2 parts:

- A *mobicore* subdirectory that groups all the Android user space components (binaries and libraries).
- And a *gud* subdirectory which contains all the Android kernel components.

2.1.1 <t-base User-space Components

The usual way to build t-base user space components is from a complete Android source tree:

Simply copy the *mobicore* folder to an Android source tree; all the components embed their own internal *makefile.mk*.

You should have the following organization:

```
Android root/
    External/
      mobicore/
        MobiCoreDriverLib/
        ProvisioningLib
        RootPA/
```

They will be automatically built within the next build of full Android source tree.

Else usual Android command can be used to build only these components:

```
$ mmm external/mobicore/
```

2.1.2 <t-base Kernel-space Components

The <t-base Linux driver directory in the <t-base package is designed to be directly dropped in the Linux kernel source tree, in the *drivers* folder.

It contains two modules:

- The <t-base Linux driver (with two interfaces for applications and <t-base Daemon).
- The <t-base Kernel API.

Once the *gud* directory has been copied, you should have the following organization:

```
Linux kernel/
    drivers/
        gud/
            Kconfig/
            Makefile/
            MobicoreDriver/
            MobicoreKernelApi/
```

Please note that dropping it in a different folder in the Linux source tree will not work.

To enable automatic building inside the kernel follow these steps:

- Update Linux kernel configuration file *linux/drivers/Kconfig* with following line (must be inserted before the last *endmenu* line):

  ```
  source "drivers/gud/Kconfig"
  ```

- And add the <t-base Linux driver to the Linux kernel build file *linux/drivers/Makefile* :

  ```
  obj-y += gud/
  ```

- Then, run *make menuconfig* and in the *Device Drivers* page of the configuration menu, please select:
  - *Linux Mobicore Support*
  - *Linux Mobicore API*

  ```
  <*> Linux Mobicore Support
  [ ]   Mobicore Module debug mode
  <*>   Linux Mobicore API
  ```

- Finally, run *make* again and the <t-base kernel components will be included in the kernel image.

2.2 INTEGRATION IN ANDROID

2.2.1 Integration in Device File System

The table below gives the directories in the Android file system where <t-base components should be placed:

Component	Name of the binary	Device directory
<t-base Daemon	`mcDriverDaemon`	`/system/bin/`
Root Provisioning Agent	`RootPA.apk`	`/system/app/`
<t-base Client library	`libMcClient.so`	`/system/lib/`
<t-base registry interface	`libMcRegistry.so`	`/system/lib/`
Provisioning library	`libgdmcprov.so`	`/system/lib/`
Curl library	`libcrul.so`	`/system/lib/`
Log library	`libcommonpawrapper.so`	`/system/lib`
Content Management Trustlets	`07010000…0000.tlbin` *(tlcm.axf)*	`/system/app/mcRegistry`

The RootPA is a standard Android app and needs to be signed like any non-system apps (version provided by Trustonic is not signed).

If the <t-base Linux driver modules are not built-in the Linux kernel, then you should have the following:

Component	Name of the binary	Device directory
<t-base user device	`mcDrvModule.ko`	`/system/lib/modules/`
<t-base kernel device	`mcKernelApi.ko`	`/system/lib/modules/`

2.2.1 Permissions and Access Rights

- <t-base daemon:
  When using Android integration Daemon should automatically get the correct permissions (0755)
- <t-base libraries:
  No specific permission is needed for libraries.
- <t-base linux driver:
  <t-base linux driver has two devices, device for user access and device for daemon access (system).

  Access permissions for these are defined by *udev* following way:

```
/dev/mobicore-user 0666 system:system
/dev/mobicore 0600 system:system
```

2.2.2 Directories Requirements

There are two directories that should be created in device file-system for <t-base. These paths are used by <t-base components at runtime.

2.2.2.1 mcRegistry directory

This directory is used by t-base to store non-permanent contents, that can be deleted with factory reset or device wipe. Usually it used to store third party trustlet content (OTA installed containers).

Mandatory path is:

```
/data/app/mcRegistry
```

and permissions should be 770 (System, Read/Write/Execute).

2.2.2.1 Persistent Trustlets and Secure Drivers

Trustlets and Secure Drivers can be stored anywhere in the device file system. However, Persistent Trustlets or Secure Drivers binaries must be placed in a persistent partition of file system, persistent among factory reset or device wipe.

For legacy reason, some System Trustlets (like the tlcm, Content Management System Trustlet) are expected to be found in

```
/system/app/mcRegistry
```

(No content generated here at runtime by t-base, only used to store legacy persistent System Trustlets)

2.2.3 Starting up <t-base

For automatically running <t-base daemon at start-up, add the following lines to the *init.rc* file of the Android device:

```
service mobicore /system/bin/mcDriverDaemon
    user system
    class main
```

Daemon has to have system permissions for user and group. Otherwise OEM is responsible to set correct parameters for the service.

OEM is also responsible to decide how the system should behave in error cases like if the start-up fails or if the daemon crashes.

3 <T-BASE SECURE WORLD COMPONENTS

Secure World components of <t-base are available in the package directory *SecureWorldIntegration*. It contains:

- The <t-base binary (secure operating system running in Secure World).
- The Content Management trusted application (System trusted application responsible for Containers management).

The <t-base images are provided in binary (Debug or Release) under *SecureIntegration/t-base/bin/MobiCore/*:

- the Debug version is much slower but prints all <t-base traces to the Linux Kernel log(dmesg)
- the Release version is very fast but there are not internal <t-base traces (note that even if you use the Release image of <t-base you can get traces from the trusted applications if you have compiled them as Debug).

The images are un-configured (mobicore*.raw.img*) and should be configured with a hash of a system trusted application public key before using them.

For reference package contains also images signed with Trustonic test keys (*.img*).

The Content Management trusted application is provided in binary (Debug and Release) under *SecureIntegration/tbase/bin/TlCm*. The trusted application is provided un-configured (*tlCm.axf*) and should be configured with the <t-base KPH request signing key and signed with the System trusted application key pair.

3.1 <T-BASE IN BOOT CHAIN

<t-base image should be started up in the boot flow as early as possible. Booting up with <t-base is a critical task in the system design and it should be done in co-operation with SOC vendor and Trustonic. There are several options for implementing <t-base boot. The purpose of this chapter is to give basic level of understanding what needs to be done.

- It is recommended to store <t-base to a permanent location in Boot partition or some other permanent storage. Usually early in the bootchain there is no normal file system present at that time.
- <t-base image must be signed. The responsibility of doing this is usually on OEM/ODM.
- <t-base image should be loaded to secure memory (internal or DRAM). Loading of <t-base image to secure memory should be done by bootloader or by Uboot.
- After loading <t-base into secure memory signature of <t-base image should be verified.
- Before calling <t-base, the caller needs to set boot configuration block for <t-base. This block is used for exchanging some predefined data between the caller and <t-base.
- Calling <t-base can usually be just a basic jump instruction. Depending on the execution environment, something else might also be needed to be done. As the initialization of <t-base returns to the caller, it means that the caller environment (bootloader/uboot) should be saved and restored for the time period execution is in TrustZone side.

If <t-base signature check or boot fails, it is up to OEM or SOC vendor to decide whether the device can be booted without <t-base or not.

3.2 TRUSTED APPLICATIONS AND DRIVERS OVERVIEW

In <t-base environment, there are 3 types of secure components:

- System trusted applications,
- Service Provider trusted applications,
- Drivers.

Service Provider trusted applications are applications deployed at runtime over the air. Service Provider trusted applications are deployed through a Trusted Service Manager (TSM) which is not covered by this document.

System trusted applications and drivers are "pre-installed" on the device. To ensure the security of these trusted applications or drivers, they need to be signed and verified at runtime by <t-base. To do this, the hash of the System trusted applications public key is inserted to <t-base raw-image.

By convention, trusted applications are named as *UUID*.tlbin and drivers as *UUID*.drbin, where UUID is a 16-byte hex value that is generated according to RFC-4122.

Any UUID's can be used by OEMs. There is no acceptance control of UUID's done by Trustonic. However, UUID's having the last 12 bytes all zero are reserved for Trustonic:

- *0000 0000 0000 0000 0000 0000 0000 0000*,
- *0000 0001 0000 0000 0000 0000 0000 0000*,
- ... ,
- *FFFF FFFF 0000 0000 0000 0000 0000 0000*.

For more details on trusted applications and drivers development, please check documentations provided in <t-sdk directories.

3.3 CONFIG AND SIGNING OF T-BASE COMPONENTS

The following section describes how to manage the different private and public keys in <t-base product to ensure the authenticity and integrity of <t-base binary and System trusted applications.

It is mandatory for the OEM to go through all the steps of this section to customize the keys involved in the different security mechanism (for testing purpose, the <t-base integrator would also have to do these steps).

3.3.1 Keys in <t-base Environnent

There are three keys involved in t-base environment that are needed to be taken care of in the integration phase:

- *PrK.Vendor.Boot* and *PuK.Vendor.Boot*:
  <t-base signing PKI key pair for boot, these keys are used for verifying integrity of <t-base image.
- *PrK.Vendor.TltSig* and *PuK.Vendor.TltSig*:
  <t-base System trusted applications signing PKI key pair, they are used for signing System trusted applications and drivers that are installed by default on the device.
- *Prk.Kph.Request* and *PuK.Kph.Request*:
  The <t-base KPH request signing key pair, it is used to verify the setting up of <t-base Content Management in production (used to verify exchanges with KPH).
  This key, only involved during productions, **is directly shared by Trustonic with the OEM** (covered by documentation *t-base_Provisioning_Manual.pdf*).

OpenSSL can be used to generate the key pairs that will be injected or used with <t-base components:

```
$ openssl genrsa -3 -out VendorBoot.pem 2048
$ openssl rsa
    -in VendorBoot.pem
    -pubout
    -outform PEM
    -out VendorBoot_pub.pem
$ openssl genrsa -3 -out VendorTltSig.pem 2048
$ openssl rsa
    -in VendorTltSig.pem
    -pubout
    -outform PEM
    -out VendorTltSig_pub.pem
```

(PEM format is expected by <t-base components).

3.3.2 Configuring <t-base Binary

3.3.2.1 Configuring <t-base Image

The first step to configure <t-base image is to inject the hash of the <t-base System trusted application signing public key (*PuK.Vendor.TltSig*) into the raw <t-base image. The goal is to ensure origin of System trusted applications and driver.

The MobiConfig tool is provided in t-base package to inject a key into a binary (*SecureIntegration/tools/MobiConfig*):

```
$ cd SecureIntegration/tools/MobiConfig
$ java -jar MobiConfig/Out/Bin/MobiConfig.jar
    -c
    -i mobicore.img.raw
    -o mobicore.img
    -k VendorTltSig_pub.pem
```

3.3.2.2 Signing the System Trusted Applications and Drivers

Then, all the System trusted applications and drivers of the system must be signed with the System trusted application Signing key.

The MobiConvert tool is provided in t-base package to sign a trusted application (*t-base-dev-kit/t-sdk/TlSdk/Out/Bin/MobiConvert*, also details in <t-sdk documentations):

```
$ cd t-base-dev-kit/t-sdk/TlSdk/Out/Bin/MobiConvert
$ MobiConvert/Out/Bin/MobiConvert.sh
    -b my_system_trustlet.axf.conf
    -servicetype 3
    -output 00010002000300040005000600070008.tlbin
    -k VendorTltSig.pem
```

3.3.2.3 Signing <t-base Image

Signing <t-base, with *VendorBoot.pem*, has to be done by OEM methods and signature check of <t-base image, before starting up, has to be ensured by the platform bootloader.

3.3.3 Configuring Content Management Trusted Application

The Content Management trusted application is the secure peer of the Normal World t-base component RootPA. It is involved:

- In production, during device manufactory, to generate the Authentication Token.
- At runtime, once the device is deployed, for Content Management on the Secure side (components creation).

3.3.3.1 Configure Content Management Trusted Application

The goal of this step is to inject the KPH request signing key (*PuK.Kph.Request*) into the Content Management trusted application so that <t-base can authenticate the requests coming from the KPH.

Each key comes with a KID to identify it (in some use cases, several keys can be supported):

```
$ java -jar MobiConfig/Out/Bin/MobiConfig.jar
    -c
    -i tlCm.axf
    -o tlCm.axf.conf
    -k MobiConfig_GuD_KPH_public_key.pem
    --kid 0
```

3.3.3.2 Signing Content Management Trusted Application

The Content Management trusted application is a System trusted application. Once configured with the KPH request signing key and its KID, it must be signed with the usual System trusted application Signing key (*PrK.Vendor.TltSig*):

```
$ MobiConvert/Out/Bin/MobiConvert.sh
    -b tlCm.axf.conf
    -servicetype 3
    -output 07010000000000000000000000000000.tlbin
    -k VendorTltSig.pem
```

The signed Content Management trusted application must finally be installed on the device in the Registry (the persistent one, in case of device wipe, the Content Management must still work).

3.4 <T-BASE BOOT PARAMETERS (BOOT INTERFACE)

The entity that starts up <t-base (bootloader or Uboot) should use <t-base boot configuration block as an interface towards <t-base. The configuration block can be customized according to needs. It should transfer at least the following information to <t-base.

Example of <t-base boot args on a Uboot integration:

Register	Values
r0	0, Cold boot. 1, Wake up from sleep.
r1	Pointer to MCSysInfo block (physical address, MCSysInfo_ptr). Defined below.
r13 / sp	MC boot stack (physical address) / 20 words available
r14 / lr	Start address of NWd. <t-base jumps to this NWD address after changing the NS bit.

Here is the definition of the <t-base system information structure:

```
typedef struct {
    uint32_t            magic;       //
    uint32_t            version;     // 0x00010000
    uint32_t            length;      // 0x00000030 (bytes)
    uint32_t            flags;       // Reserved
    struct mem_info_t   dram_total;  // Total DRAM area
    struct mem_info_t   dram_sec;    // Secure DRAM area
    struct mem_info_t   sram_total;  // Total SRAM area
    struct mem_info_t   sram_sec;    // Secure sRAM area
} MCSysInfo_t, *MCSysInfo_ptr;

typedef struct {
    uint32_t       base; // physical address (32bits)
    uint32_t       size;
} mem_info_t;
```

4 VALIDATION TEST SUITE

4.1 OVERVIEW

The <t-base Validation Test Suite is the test suite Trustonic provides to OEMs. You can find the source code, binaries and trusted application in the *ValidationTestSuite* folder.

- A test specification is provided in sub-folder */Documentation/ValidationTestSuite.tgz*. The ZIP file contains PDF and HTML documents of the test cases.
- Source Code is located in *./Locals/Code/*
- Trusted Application (and Drivers) needed are contained in folder *./Trustlets* as Debug '.axf' files. They need to be signed by MobiConvert tool which then creates the '.tlbin' files.
- The *ValidationTestSuite* binary is located in *./Out/Bin/* as Debug version.

4.2 BUILDING THE TEST SUITE

The full source code of the Validation Suite and the required components are located in the release package: *ValidationTestSuite/Application/Locals/Code*

Before launching the helper build script, please make sure you have modified the common configuration script *setupDriver.sh*:

```
$ cd Application/
$ ./build.sh
The build environment is not set!
Trying to source setupDrivers.sh automatically!
Gdbserver       : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup        : libs/armeabi/gdb.setup
Compile++ thumb  : ValidationTestSuite <= McDriverTest.cpp
Compile++ thumb  : ValidationTestSuite <= mcErrorStrings.cpp
Compile++ thumb  : ValidationTestSuite <= TestUtils.cpp
Compile++ thumb  : ValidationTestSuite <= 00100_mcOpenDevice.cpp
Compile++ thumb  : ValidationTestSuite <= 00200_mcCloseDevice.cpp
Compile++ thumb  : ValidationTestSuite <= 00300_mcMallocWsm.cpp
Compile++ thumb  : ValidationTestSuite <= 00400_mcFreeWsm.cpp
Compile++ thumb  : ValidationTestSuite <= 00500_mcOpenSession.cpp
Compile++ thumb  : ValidationTestSuite <= 00600_mcCloseSession.cpp
Compile++ thumb  : ValidationTestSuite <= 00700_mcNotify.cpp
Compile++ thumb  : ValidationTestSuite <= 00800_mcWaitNotification.cpp
```

```
Compile++ thumb  : ValidationTestSuite <=
                   00900_mcGetSessionErrorCode.cpp
Compile++ thumb  : ValidationTestSuite <= 01000_mcMap.cpp
Compile++ thumb  : ValidationTestSuite <= 01100_mcUnmap.cpp
Compile++ thumb  : ValidationTestSuite <= 00100_mcMapBoundaryTests.cpp
Compile++ thumb  : ValidationTestSuite <= 00200_Tci_tests.cpp
Compile++ thumb  : ValidationTestSuite <= 00100_TrustletCommands.cpp
Compile++ thumb  : ValidationTestSuite <= 00300_Dead_Trustlet.cpp
Compile++ thumb  : ValidationTestSuite <=
                   00301_Trustlet_Endless_Loop.cpp
Compile++ thumb  : ValidationTestSuite <= 00400_Communication.cpp
Compile++ thumb  : ValidationTestSuite <= 00500_DynamicDriver.cpp
Compile++ thumb  : ValidationTestSuite <= 00501_DynamicDriver2.cpp
Compile++ thumb  : ValidationTestSuite <= 00502_DynamicDriver3.cpp
Compile++ thumb  : ValidationTestSuite <= 00503_DynamicDriver4.cpp
Prebuilt        : libCppUTest.a <= ../CppUTest/Bin/Debug/
Prebuilt        : libstdc++.a <= <NDK>/sources/cxx-stl/gnu-
                  libstdc++/libs/armeabi/
Prebuilt        : libMcClient.so <=
                  ../../MobiCoreDriverLib/Out/Bin/Generic/Debug/
Executable      : ValidationTestSuite
Install         : ValidationTestSuite =>
                  libs/armeabi/ValidationTestSuite
```

Now the compiled ValidationTestSuite is rebuilt in:

ValidationTestSuite/Application/Locals/Code/libs/armeabi

4.3 INSTALLATION

Please make sure you have already installed the <t-base Linux drivers and Daemon and that they are running.

The trusted applications in the release are only available as '.axf' files. They need to be signed by MobiConvert tool which then creates the '.tlbin' files:

```
$ cd ValidationTestSuite/Trustlets
$ java -jar ../../MobiConvert/Out/Bin/MobiConvert.jar
    -bin 05010000000000000000000000000000.axf
    -servicetype 3
    -i 2
    -output 05010000000000000000000000000000.tlbin
    -keyfile VendorTltSig.pem

$ java -jar ../../MobiConvert/Out/Bin/MobiConvert.jar
    -bin 02040000000000000000000000000000.axf
    -servicetype 1
    -output 02040000000000000000000000000000.drbin
    -d 1024
    -iv 1.0
    -keyfile VendorTltSig.pem

$ cp 02040000000000000000000000000000.drbin
     02040000000000000000000000000000.tlbin
```

This will result in 3 new binaries:

1. 05010000000000000000000000000000.tlbin - test trusted application
2. 02040000000000000000000000000000.drbin – generic <t-base driver
3. 02040000000000000000000000000000.tlbin - generic <t-base driver/ trusted application

Copy all the signed trusted application binaries to the device in the registry (by default */data/app/mcRegistry*) and launch the *ValidationTestSuite* binary on the device.

4.4 RUNNING THE TEST SUITE

The <t-base Validation Test Suite supports command line arguments are:

```
$ ./ValidationTestSuite --help
*** Build by ozengin@ukko, svn:zenginon@r14343 ###
Options:
-v Verbose, print each test name as it runs
-r <times> Repeat the tests some number of times
-g <group> Only run tests whose group contains the substring group
-n <name> Only run tests whose name contains the substring name
-e <group> Exclude tests whose group contains the substring group
-x <name> Exclude tests whose name contains the substring name
-o {normal|junit|csv} Log output format
```

To execute the entire <t-base Validation Test Suite, you have to go in the android shell:

```
$ adb shell
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/data/app
$ cd /data/app $ ./ValidationTestSuite -v
EXECUTING McDrvApi_00100_mcOpenDevice_GC00100_IdDefault - 3 ms
EXECUTING McDrvApi_00100_mcOpenDevice_BC00101_InvalidDevice1 - 4 ms
EXECUTING McDrvApi_00100_mcOpenDevice_BC00102_InvalidDevice2 - 3 ms
EXECUTING McDrvApi_00100_mcOpenDevice_BC00104_OpenRepeatedly - 3 ms
EXECUTING McDrvApi_00200_mcCloseDevice_GC00100_IdDefault - 2 ms
...
OK (124 tests, 124 ran, 9105233 checks, 0 ignored, 0 filtered out,
157205 ms)
```

It’s also possible to run individual tests or sub groups of the *ValidationTestSuite*. As you can see above each line is made up of a test name and a test group.

For example here, *McDrvApi_00300_mcMallocWsm_BC00105_WsmNullPointer*:

- *00300_mcMallocWsm* is the test group
- *BC00105_WsmNullPointer* is the test name.

If you would like to run only this test you have to type:

```
$ ./ValidationTestSuite -n BC00105_WsmNullPointer
*** Build by ozengin@ukko, svn:zenginon@r14343 ###
.
OK (124 tests, 1 ran, 2 checks, 0 ignored, 123 filtered out, 9 ms)
```

But if you want to run the whole group, you execute:

```
./ValidationTestSuite -v -g 00300_mcMallocWsm
*** Build by ozengin@ukko, svn:zenginon@r14343 ###
EXECUTING McDrvApi_00300_mcMallocWsm_GC00100_IdDefault4k - 32 ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00101_InvalidDevice1 - 2 ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00101_InvalidDevice4 - 2 ms
EXECUTING  McDrvApi_00300_mcMallocWsm_BC00102_InvalidDeviceMinus1  -  2
ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00103_Len0 - 2 ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00104_LenMinus1 - 2 ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00105_WsmNullPointer - 2 ms
EXECUTING McDrvApi_00300_mcMallocWsm_BC00106_ClosedDevice - 2 ms
EXECUTING
McDrvApi_00300_mcMallocWsm_GC00107_StresstestMultipleAllocations
Stress-test: mcMallocWsm called 16 times, allocated 15 chunks.
- 562 ms
EXECUTING McDrvApi_00300_mcMallocWsm_GC00108_StresstestMultipleSizes
Stress-test: Calling mcMallocWsm with size 4096 == 0x1000. ok.
Stress-test: Calling mcMallocWsm with size 8192 == 0x2000. ok.
Stress-test: Calling mcMallocWsm with size 16384 == 0x4000. ok.
Stress-test: Calling mcMallocWsm with size 32768 == 0x8000. ok.
Stress-test: Calling mcMallocWsm with size 65536 == 0x10000. ok.
Stress-test: Calling mcMallocWsm with size 131072 == 0x20000. ok.
Stress-test: Calling mcMallocWsm with size 262144 == 0x40000. ok.
Stress-test: Calling mcMallocWsm with size 524288 == 0x80000. ok.
Stress-test: Calling mcMallocWsm with size 1048576 == 0x100000. ok.
Stress-test: mcMallocWsm called 9 times,
Biggest allocated chunk: 1048576 byte.
- 228 ms
OK (124 tests, 10 ran, 44 checks, 0 ignored, 114 filtered out, 844 ms)
```