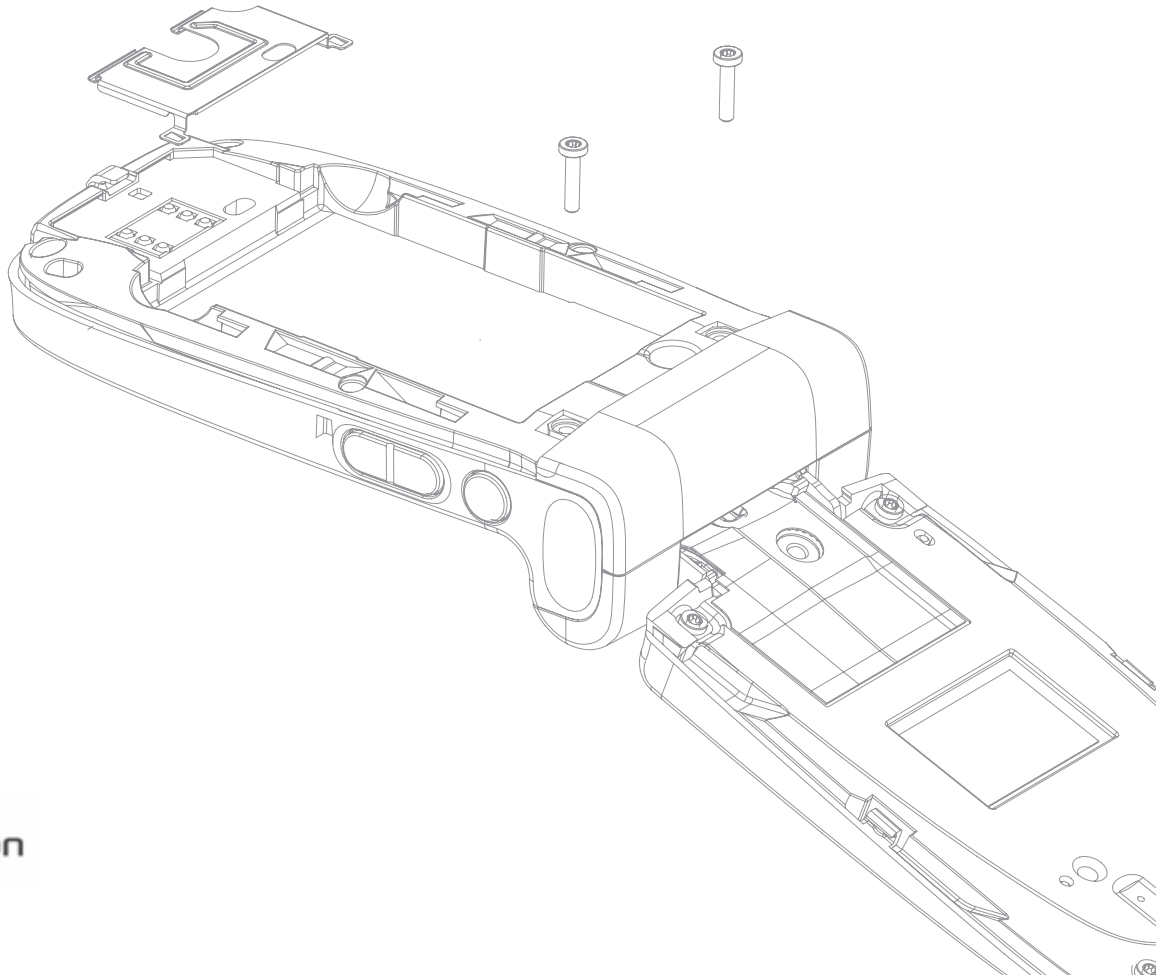
Developers guidelines

DEVELOPER WORLD THE FAST TRACK FROM MIND TO MARKET

October 2006

Signing applications

for Sony Ericsson UIQ 3 phones



Sony Ericsson

Preface

Purpose of this document

This document describes how to handle signing of native Symbian™ OS v9 applications for Sony Ericsson phones. The document is intended for developers of UIQ™ 3 C++ applications who want insight in the implications of Symbian Platform Security (PlatSec) on the deployment and installation of applications in these phones.

Readers who will benefit from this document include support engineers and software developers.

It is assumed that the reader is familiar with C++ program development and deployment in general.

These Developers guidelines are published by:

Sony Ericsson Mobile Communications AB,
SE-221 88 Lund, Sweden

Phone: +46 46 19 40 00
Fax: +46 46 19 41 00
www.sonyericsson.com/

Second edition (October 2006)
Publication number: EN/LZT 108 8297 R2A

Sony Ericsson Developer World

On www.sonyericsson.com/developer, developers will find documentation and tools such as phone White Papers, Developers Guidelines for different technologies, SDKs and relevant APIs. The website also contains discussion forums monitored by the Sony Ericsson Developer Support team, an extensive Knowledge Base, Tips & Tricks, example code and news.

Sony Ericsson also offers technical support services to professional developers. For more information about these professional services, visit the Sony Ericsson Developer World website.

Document conventions

Products

Sony Ericsson mobile phones are referred to in this document using generic names as follows:

Generic names Series	**Sony Ericsson mobile phones**
P990	P990i, P990c
M600	M600i, M600c
W950	W950i, W950c
W958	W958c

Abbreviations

ACS	Advanced Cryptographic System
API	Application Programming Interface
DRM	Digital Rights Management
HAL	Hardware Abstraction Layer
IMEI	International Machine Equipment Identity
MMP	File extension for the Project Definition File
SDK	Software Development Kit
USB	Universal Serial Bus

Typographical conventions

Code is written in Courier font, for example: `TInt CCamera::CamerasAvailable()`

Trademarks and acknowledgements

Document history

Change history		
2006-05-10	Version R1A	First version published on Developer World
2006-10-03	Version R2A	Updated with reference to W958c

Contents

Symbian OS v9 security architecture

This chapter gives a general overview of the Symbian OS v9 security features as implemented in Sony Ericsson mobile phones.

Introduction

Symbian OS version 9.x is specifically intended for mid-range phones to be produced in large numbers of units. The open development platform, featuring many new key technologies, offers large opportunities for ISVs (Independent Software Vendors) to find markets for their products.

Introduction of new functionality, such as DRM (Digital Rights Management), Device Management and enhanced networking functionality, has required changing of the Symbian OS core to support vital security concepts such as data protection or “caging” and restricting usage of some “sensitive” APIs.

Symbian OS v9 Platform Security (PlatSec) has been enhanced to provide a high degree of protection against malicious or badly implemented programs, which means that such programs are efficiently detected and prevented from executing on the platform. On the other hand, applications that have been tested and found “trustworthy”, can gain authorization to be installed and executed on the platform, without further security confirmations on the user level. This authorization is done via the Symbian Signed programme which include procedures for signing of applications using certificates, both in the development phase and when the application is to be packaged and distributed to the market.

This document is primarily intended to guide Symbian OS v9 application developers in the process of creating applications to be authorized via the Symbian Signed programme.

Capabilities

The term “capability” has been introduced with Symbian OS v9 Platform Security. A capability can be assigned to a program, guaranteeing that the process started by the program uses the associated Symbian OS v9 functionality (for example an API) in a safe way. Thus, a capability can be regarded as a granted protection of its associated APIs. The protection is granted either by a digital signature, or by a user permission given for an unsigned application at installation.

An application can be signed at different levels of trust. The higher level of trust, the more sensitive capabilities can be granted access. Capabilities are therefore grouped into four different sets, each applicable for a certain level of trust. For more information, see “Capability mapping” on page 11.

Restricted and unrestricted APIs

A majority of Symbian APIs are classified as “unrestricted”, which means that they require no authorization, since they have no harmful security implications on the device or network integrity. Unrestricted APIs are not associated with capabilities, since no protection is needed.

APIs with potential security implications are referred to as “restricted”. Restricted APIs are grouped into capabilities based on their functionality. Applications are granted access to capabilities rather than to APIs in order to simplify the process of authorization.

Identifiers

Symbian OS v9 Platform Security also requires that applications can be uniquely identified and strictly classified to reflect their PlatSec level of trust. For example, signed and unsigned application are clearly separated by having UID values in separated value ranges.

Unique Identifiers, UIDs

In Symbian OS, objects are identified by three 32 bit globally unique identifiers, referred to as UID1, UID2 and UID3.

- UID1 is a system level identifier, distinguishing for example executables, DLLs and file stores.
- UID2 distinguishes objects with the same UID1 based on different interfaces. For example GUI applications have a common UID2 value.
- UID3 can be seen as a project identifier, for example, all objects belonging to a given program may share a UID3 value.

With Symbian OS v9, allocation of UID3 values has been changed to further enhance security, for example to implement the data caging feature. Applications developed for public distribution must be assigned a globally unique UID3 value, which is utilized through an automated UID allocation system implemented within the Symbian Signed programme.

Allowed UID values have been split into one protected range for signed application and one unprotected range for unsigned applications. Only signed applications can use UIDs in the protected range, and only protected range UID values are allowed for signed applications. This is validated in the Symbian Signed process. On the other hand unsigned applications are not allowed to use UIDs in the protected range, and can only be installed with a UID from the unprotected range of values.

Secure Identifier, SID

Symbian OS v9 applications are assigned a SID value, which is automatically set to the UID3 value, unless explicitly specified by the developer. The SID value determines the name of the folder where private application data is stored.

The SID value can be specified in the .MMP file of the application, but this option should only be used in special cases. Normally the automatically set value of UID3 should be accepted.

Vendor Identifier, VID

A Vendor ID can be used at runtime to identify the source of the binary. It is mainly of interest for phone manufacturers and network operators, for example when needing to restrict access to a certain service to applications from specific vendors. Most developers have no need for a VID, and the default VID value (0) can then be used.

IF a VID value other than 0 is to be used, it is specified in the .MMP file of the application. VID values must not be specified for unsigned applications.

Data caging

Data caging has been introduced in Symbian OS v9 to prevent one application to overwrite data belonging to another application.

The file system has the following structure:

- `\sys` : This is the restricted system area which is only accessible for highly trusted system processes.
- `\sys\bin` : Holds all executables such as EXEs, DLLs and plug-ins.
- `\private` : Each application has its own private view of the file system consisting of `\private\<SID>\`. This folder is only accessible by the application itself, the software installation program and applications trusted with capabilities on the highest level (granted by the phone manufacturer).
- `\resource` : A public, read-only directory allowing files to be publicly shared without compromising integrity. An application should, for example, put its UI resource files and icon files in `\resource\apps`.

Other directories are public and can be read from or written to by any program.

Unsigned - sandboxed applications

Unsigned applications are applications that have not been authorized through any signing process. Unsigned applications are allowed access to all unrestricted APIs and a small number of restricted APIs. Such applications are often referred to as “Unsigned - Sandboxed”, which implies that they have access to a limited number of APIs (the sandbox).

Unsigned - sandboxed applications using any of the restricted APIs, still need to be authorized by the user at install time. When the application is installed on the phone, the user is prompted to accept that the application is granted “blanket” permissions to any functions that it requires. If the user accepts, the application is granted permission to the functions as long as it is installed in the phone. If the user rejects, the installation is aborted.

Some capabilities can only be granted “one-shot” permissions when assigned to an unsigned application. Every time the application needs access to one of these capabilities, the user is prompted to accept the action that the application is about to perform. If the user rejects, an error condition is raised, which have to be managed by the code.

The following table lists allowed user granted permissions per capability for unsigned applications:

Capability	User granted permission
NetworkServices	One-shot
LocalServices	Blanket
ReadUserData	One-shot
WriteUserData	One-shot
UserEnvironment	Blanket
Location	One-shot

Note: An application that could be deployed as an unsigned - sandboxed application may as well be subdued to the Symbian Signed process. When an application like this has been signed, no user interaction is required at installation, and the mapped capabilities are automatically granted blanket permissions.

Note: Sony Ericsson strongly recommends users only to install signed applications in their phones and only allows signed applications to be distributed through its official sales channels, thus encouraging developers who want to market their applications for wide use with Sony Ericsson phones, always to favour signed applications before unsigned.

Symbian Signed applications

The security enhancements of Symbian OS v9, have enforced a number of changes in the Symbian Signed process. As a consequence, also developer procedures for having applications Symbian Signed have changed considerably.

Capability mapping

Capabilities are categorized into three separate sets on different levels, depending on their potential impact on the device, the network or the user. The more serious impact a capability might have, the higher level of trust is required by an application to access it, and the more testing is needed in the Symbian Signed process to make sure that the application makes use of the capability in a safe and secure manner.

Note that unrestricted APIs have no capabilities associated with them. As mentioned above, they can be included even in unsigned applications and are automatically given blanket permission at installation.

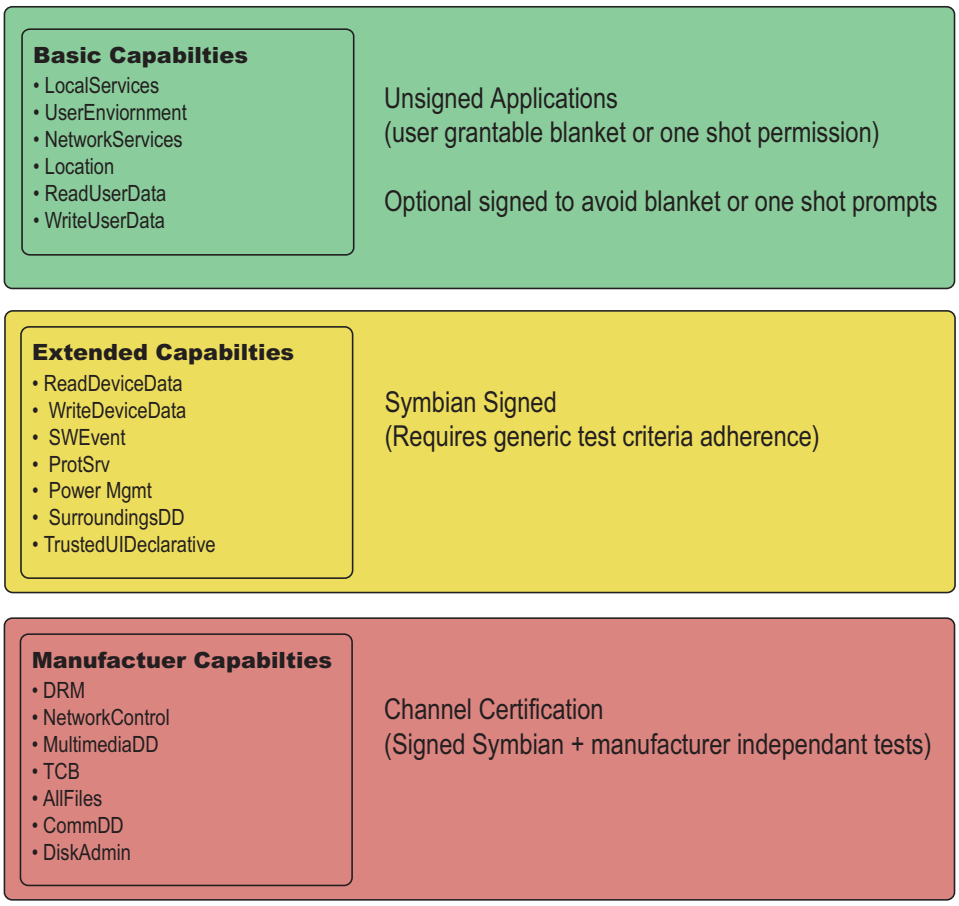
The three cabability sets are:

- Basic capabilities
  Applications requiring basic capabilities can either be Symbian Signed or unsigned. When installing an unsigned application requiring one or more of the basic capabilities, the user is prompted to grant

blanket or one-shot permissions at install time. Only standard (generic) testing is required for an application to be Symbian Signed.

- Extended capabilities
  Highly trusted applications may be granted access to this set. For an application utilizing one or more capabilities in this set to be Symbian Signed, it has to go comply to extended testing criteria. The developer of the application must also explicitly declare which APIs of the capability are used, and why they are needed.

- Phone manufacturer approved capabilities
  The highest level of trust is required for applications that may have an impact on the functionality of the device. The only way for an application to have access to these capabilities is through a “channeled” signing procedure which involves approval by Sony Ericsson.



Access rights to capabilities are cumulative, for example, an application signed with the basic set is also granted access to all the unrestricted APIs.

An application signed for a particular set is not granted access to all capabilites of the set. The installer gives permissions only to those capabilities that the application actually requires.

Developer certificates

As a consequence of the Symbian OS v9 enhanced platform security, applications that require access to restricted APIs can not be installed on targeted devices before they have been signed, which in turn makes it impossible to test applications on real mobile phones during the development process. To take care of this, special developer certificates can be achieved via the Symbian Signed programme. Developer certificates are valid for a number of of phones, identified by their IMEI number, and allow developers to sign applications temporarily during development for installation and testing on real devices.

Testing on emulators does not normally require developer certificates.

ACS publisher ID

An application using capabilities in the extended or phone manufacturer approved sets, must be signed with an ACS publisher ID certificate when a developer certificate is requested and when the finished application is sent to be Symbian Signed. The ACS publisher ID is used to verify the identity of the developer and to certify that the application has not been modified during upload to the Symbian Signed portal. An ACS publisher ID is issued by Verisign and is valid for one year. It can be used to sign an unlimited number of applications during that period. More information on how to acquire an ACS publisher ID and the costs for it, can be found on http://www.verisign.com/products-services/security-services/code-signing/symbian-content-signing/index.html

Publisher certifiers

ACS publisher IDs are only available to recognized organizations, for example registered companies. Developers who are not qualified for an ACS publisher ID of their own, can use an alternate route for having their applications signed. Publisher certifiers are organizations having the right to certify applications for themselves or others, using their own ACS publisher ID. They are certified by Symbian to perform application testing and finally make the application Symbian Signed. Normally this procedure is connected with agreement between the developer and publisher that the application is to be distributed through the sales channel of the publisher.

Signing freeware applications

Freeware applications can also be Symbian Signed via a procedure similar to the one described above for publisher certifiers. One condition is that the application actually is freeware, that is, it is distributed and can be used without any costs for the user. It is also required that the application displays a special freeware disclaimer at installation.

Details about the preconditions for freeware signing and the process for it, can be found on https://www.symbiansigned.com/app/page/freeware.

Symbian OS v9 application signing

This chapter describes the practical implications of Symbian OS v9 platform security and the steps developers need to take during development of Symbian Signed applications.

Planning for development

There are a number of considerations to take in the beginning of the development process for a Symbian OS applications. Apart from the normal system analysis and design, also the design implications on signing requirements and testing procedures specific for the Symbian OS v9 platform must be taken into account.

Signing or not

As mentioned above, many applications do not require any capabilities and thus can be installed and executed in the “unsigned – sandboxed” environment. This may be an alternative for applications intended for personal use or for limited groups of users. For most commercial purposes and for widely spread applications, taking the costs and extra work needed for going through the Symbian Signed processes gives several advantages and is therefore highly recommended also for applications that do not require signing.

Symbian Signed advantages:

- Symbian Signed applications have passed a number of tests, which guarantees a certain quality level, giving added value for customers.
- Symbian Signed applications are made visible to Symbian partners, like network operators, distributors, phone manufacturers, and so on, via a special catalog, thus opening up a marketing channel for developers making use of the programme.
- Sony Ericsson, Nokia and several major operators and service providers, only allow applications that have passed the Symbian Signed programme to be exposed via their application shops.

The Symbian Signed programme requires that developers of applications to be submitted for signing have an ACS publisher ID from Verisign. Under certain conditions an ACS publisher ID is also required for a developer to retrieve a developer certificate for application testing on real devices. Since it may take some time to be approved for it, an ACS publisher ID should be requested in good time before it is actually required.

Required capabilities

To avoid some of the extra work and additional costs for using extended or phone manufacturer capabilities, it is a good idea always to analyze carefully which capabilities are necessary for the application. If it is possible to implement a needed functionality using an API of a lower level capability set, or an unrestricted API, this should always be preferred.

Another reason for early planning of which capabilities are needed for an application, is that a developer certificate might be required for testing on real devices. To make testing as realistic as possible, the developer certificate should grant access to the same set of APIs as the finished application. For applications using only unrestricted APIs, developer certificates are not required.

Note that an application can only be granted rights to exactly the capabilities in a set that it actually requires. When sending an application for signing, all requested capabilities must be declared for the application to be approved.The following tables list all capabilities and describe in general terms what functionalities each capability may grant to applications

.

Basic capabilities	
LocalServices	
Grants access to the local network. Applications with this capability can normally send or receive information through USB, Infrared and point-to-point Bluetooth profiles, but they can not use IP, routable Bluetooth profiles, or the phone number dialling functionalities.	
UserEnvironment	
Grants access to live confidential user information and the immediate environment of the user. Protects privacy of the user.	
NetworkServices	
Grants access to protocols capable of routing data beyond the device and its immediate personal and or local network, for example IP and GSM. This capability is required for access to the phone network, allowing, for example, dialling a phone number or sending a text message.	
Location	
Grants access to the device location.	
ReadUserData	
Grants read access to user data. System servers and application engines are free to grant this level of restriction to their data.	
WriteUserData	
Grants write access to user data. System servers and application engines are free to grant this level of restriction to their data.	

Extended capabilities	
ReadDeviceData	
Grants read access to confidential system data. System data that is not confidential does not need to be protected by this capability.	API examples: COMMS

WriteDeviceData	
Grants write access to sensitive system data.	
SWEvent	
Grants read access to confidential system data. System data that is not confidential does not need to be protected by this capability.	API examples: Test utilities, FEP
ProtServ	
Grants the right to a server to register with a protected name. Protected names begins with a "!". The kernel will prevent servers without this capability from using such a name, and will therefore prevent impersonation of protected servers.	Mainly granted to system servers.
PowerMgmt	
Grants the right to kill any process in the system, to power off unused periferals, to put the device into standby state and wake it up again, or power it down completely. Note however, that this does not control anything that might drain battery power.	API examples: WSERV
SurroundingsDD	
Grants access to logical device drivers providing input information about the device surroundings.	
TrustedUI	
Grants the right to create a trusted UI session, and thereby display dialogs in a secure UI environment.	Mainly granted to SWInstall and token servers.

Phone manufacturer capabilities	
DRM	
Grants access to protected content subject to DRM rights restrictions.	
NetworkControl	
Grants the right to modify or access network protocol controls.	
MultimediaDD	
Controls access to all multimedia device drivers, audio, camera, and so on.	API examples: MMF, ICL, ECam
TCB	

Trusted Computing Base. Grants write access to /sys and /resource.	API examples: Kernel, F32, SWInstall server
AllFiles	
Makes all files visible. Grants extra write access to files under /private.	Mainly granted to test utilities and backup & restore API examples: F32, SWInstall
CommDD	
Grants access to all communication device drivers, for example EComm and USB device drivers.	API examples: COMMS
DiskAdmin	
Grants access to some disk administration operations, for example reformatting a disk.	API examples: Shell

A complete list of functions per capability can be found in the appendix of this document, see “Functions listed by capability” on page 29, in the Symbian developer library, http://www.symbian.com/developer/techlib/v9.1docs/index.asp, and in the UIQ 3 SDK documentation.

For examples on how to avoid using restricted APIs, specifically in the phone manufacturer capability set, please refer to the following article on the Tips and tricks section on Sony Ericsson Developer World: http://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/symbian/p_avoid_restricted_apis.jsp

Creating a Symbian Signed application

In the process of developing a Symbian Signed application, several steps are connected with signing. These steps are described in short here. For more detailed information, please refer to information found in the Symbian Signed web portal, www.symbiansigned.com.

General signing procedure

Before installing an application for testing with a developer certificate, the application must be signed with that certificate. When sending a completed application to be Symbian Signed, the application package must be signed with a valid ACS publisher ID certificate. In both situations the same general procedure can be applied.

Only one extra step is required to make a signed .SIS file, compared to making an unsigned .SIS.

In the `.pkg` file, a line with the following syntax is added after the file header part and before the list of files to install.

```
*"<path>\<My_Private_Key>.key", "<path>\<My_Cert>.cer"
[,KEY="<My_PrivateKey_Pwd>]
```

for example,

```
*"files\devcert2.key","files\devcert2.cer",KEY="password"
```

MakeSis is run with the modified .pkg file create the .SIS file prepared for signing, and finally SignSis does the signing.

An alternate signing method is to omit the extra line in the .pkg file, create the .SIS file with MakeSis and finally use SignSis with parameters for the signing keys, for example:

```
signsis ?s app.sis app_signed.sis acs_id.cer private.key
```

Symbian Signed portal account registration

Most of the procedures involving the Symbian Signed portal requires login to a developer account giving access to all the services needed during the development of a Symbian Signed application. For example, requests of UIDs and Developer Certificates requires access to the account. Also downloads of Developer certificates, signed applications, and so on, are performed from account specific pages in the portal.

To register a Symbian Signed portal account:

1. Go to the portal www.symbiansigned.com.
2. Click the link "Register" in the left navigation bar.
3. Follow the instructions on the registration page.

UID allocation

For all Symbian OS v9 applications, signed or unsigned, a UID must be allocated. The UID must uniquely identify the application within the "world" where it is to be used. For example an application to be used only in one phone, requires the UID to be unique among all other applications on that phone, and an application to be sold on a global market must have a UID that is globally unique. To make sure that all allocated UIDs can be globally unique, Symbian have a system through which all developers can retrieve UIDs for all their applications. This system is available via the Symbian Signed portal.

In Symbian OS v9, UID ranges have been changed compared to earlier OS versions. The following table lists UID ranges to be allocated Symbian APP, EXE or DLL files of different categories. UID classes 0-9 (range 0x00000000 – 0x9FFFFFFF) are referred to as the *protected range*, and classes A-F (range 0xA0000000 – 0xFFFFFFFF) as the *unprotected range*. UID ranges not in this table are reserved.

UID class	UID range	Purpose
0	0x00000000 - 0x0FFFFFFF	Development use only. Normally this range is not used for Symbian OS v9 applications, since Developer certificates are required. UID classes A or E can be used during development and for test purposes. The range 0x01000000 - 0x0FFFFFFF is reserved for test purposes with pre-v9 applications.
1	0x10000000 - 0x1FFFFFFF	Legacy UID allocations. This is the UID range used for both signed and unsigned pre-V9 Symbian OS applications, not to be installed in phones with Symbian OS v9 or later.
2	0x20000000 - 0x2FFFFFFF	V9 protected UID allocations. UIDs in this range are allocated to signed Symbian OS v9 applications via the Symbian Signed portal. This requires login to the developer account.
7	0x70000000 - 0x7FFFFFFF	Vendor IDs. Normally only used by phone manufacturers and network operators. If another developer needs a VID, it can not be retrieved directly via the Symbian Signed portal, but requests can be sent via email to symbiansigned@symbian.com.
A	0xA0000000 - 0xAFFFFFFF	V9 unprotected UID allocations. UIDs in this range are allocated to unsigned applications via the Symbian Signed portal. UIDs are retrieved via login to the developer account in the portal. UIDs in this range can also be used for test applications and SDK code examples, particularly this is useful for test projects, for example games.
E	0xE0000000 - 0xEFFFFFFF	Development use only. This range is particularly suited for illustrative/learning purposes, applications that are not to be redistributed via a .SIS file, for example HelloWorld. UIDs can be randomly selected from the range as long as they do not conflict with other UIDs on the device.
F	0xF0000000 - 0xFFFFFFFF	Legacy UID compatibility range. Used for pre-v9 unsigned applications that are to be installed on a v9 device. A UID in the range 0x10000000 - 0x1FFFFFFF allocated by Symbian to a pre-v9 application can be converted to one valid for v9 by simply exchanging the first “1” to “F”. For example, a UID of 0x10034FD3 should be changed to 0xF0034FD3.

To retrieve a UID from the Symbian Signed Portal:

1. Login to your account on www.symbiansigned.com.
2. Click the link “Request UIDs” in the left navigation bar.
3. Follow the instructions on the UID request page.

When logged in on the Symbian Signed portal, UIDs allocated to the account can be viewed by clicking "View UIDs" in the left navigation bar.

ACS publisher ID

Having an application Symbian Signed requires an ACS publisher ID, either owned by the developer or by a publisher certifier. The ACS publisher ID is used to verify the identity of the developer sending the application for signing, and to create the certificate and key necessary for a secure transfer of the application into the Symbian Signed process.

To aquire a developer certificate for testing, an ACS publisher ID is required only if the application requires certain capabilities or if testing is to be done on more than one phone. However, having an ACS publisher ID simplifies the procedures for retrieving the developer certificate.

An ACS publisher ID has to be purchased from Verisign. The following is an overview of the procedures for achieving an ACS publisher ID.

1. Go to http://www.verisign.com/products-services/security-services/code-signing/symbian-content-signing/index.html and start the enrolment process by clicking the "Buy Now" button.

2. Enter the requred information in the form and accept the subscriber agreement. Verisign confirms that the enrolment process is ongoing via email.

3. When Verisign has gone through the necessary authentication and verification procedures and approved the enrolment, this is confirmed via another email.

4. Once approved, the ACS publisher ID can be picked up from the URL given in the approval email. To start the transfer, a PIN code delivered with the approval email and the challenge phrase given by the developer on enrolment, have to be entered in the browser window.

5. The retrieved ID is stored in the browser certificate store. To use the ACS publisher ID certificate for signing, it first has to be exported from the store. A special tool, that can be downloaded from Verisign, is then used to create the certificate (.cer) and private key (.key) files to be used for signing.

For more detailed information about the purchase process and the steps needed to make use of the ACS publisher ID certificate for signing Symbian applications and request developer certificates, please refer to the instructions at http://www.verisign.com/products-services/security-services/code-signing/symbian-content-signing/index.html.

Developer certificates

During the initial phases of development, applications are preferrably tested in the emulator, normally set to ignore PlatSec security restrictions. Later during development, testing on real phones is required to verify the application behavior in the targeted environment.

An application using only unrestricted APIs or user granted capabilities can be tested on real phones without being signed. An application requiring any other capability has to be signed with a developer certificate to allow it to be installed for testing on real Symbian OS phones.

Developer certificates can be issued on three different levels, depending primarily on wich capabilities are required and how many specific phones (identified by their IMEI) are to be used for testing. The following table lists granted capabilites for each certificate level, and the requirements for a developer to be allowed to request a developer certificate on a certain level.

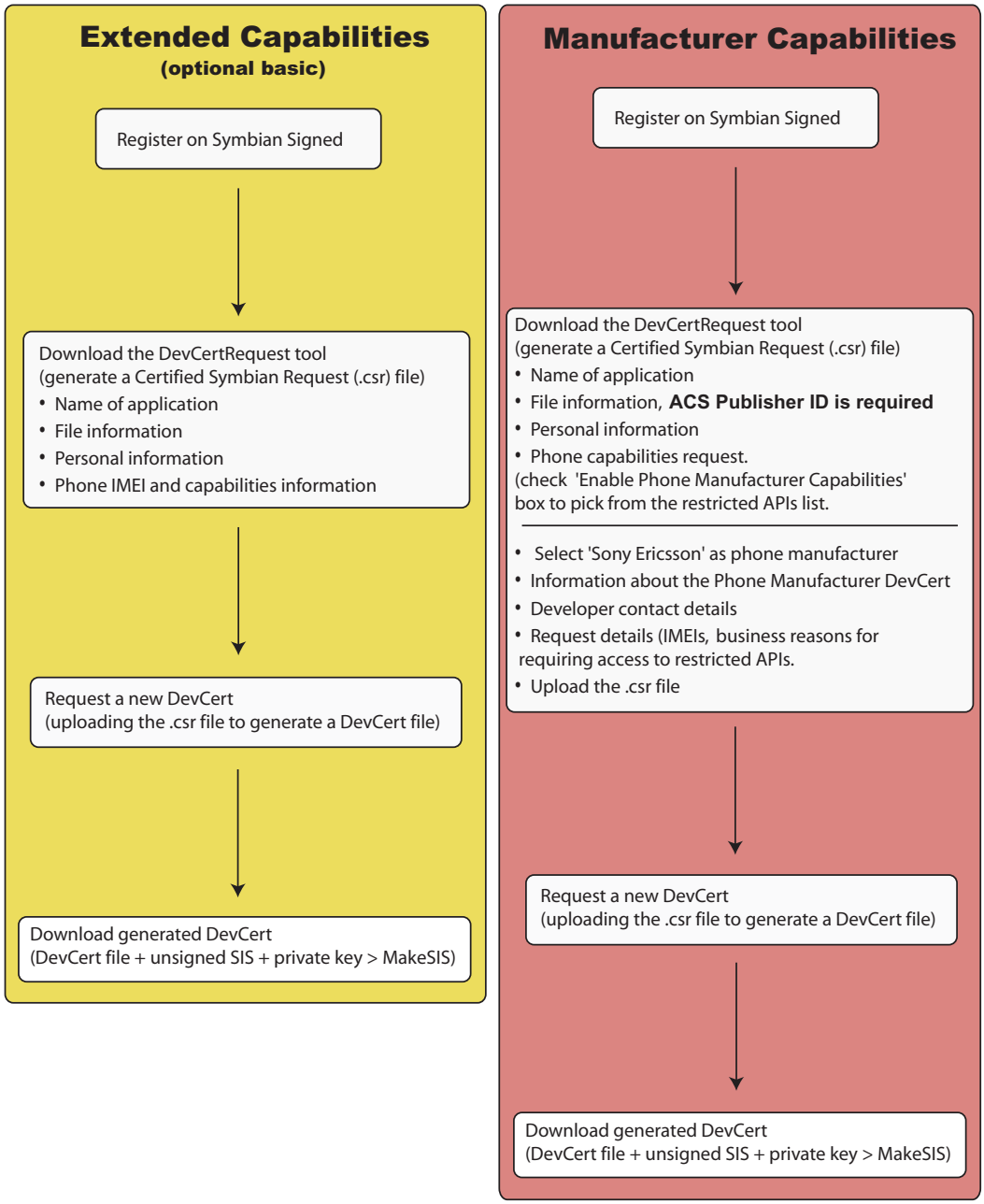
Number of IMEIs	1	1-20	Sony Ericsson approved
Identity requirements	• Registration • IMEI	• Registration • IMEIs • ACS Publisher ID	• Registration • IMEIs • ACS Publisher ID • Sony Ericsson approval
LocalServices	•	•	•
Location	•	•	•
NetworkServices	•	•	•
PowerMgmt	•	•	•
ProtServ	•	•	•
ReadUserData	•	•	•
SurroundingsDD	•	•	•
SWEvent	•	•	•
UserEnvironment	•	•	•
WriteUserData	•	•	•
ReadDeviceData		•	•
TrustedUI		•	•
WriteDeviceData		•	•
AllFiles			•
CommDD			•
DiskAdmin			•
DRM			•
MultimediaDD			•
NetworkControl			•
TCB			•

Note that this mapping is not fully conformant to the capabiliy sets (basic, extended and phone manufacturer approved) as defined in Symbian OS.

Requesting a developer certificate

Developer certificates can only be retrieved via the Symbian Signed portal by registered users. The process for retrieving a Sony Ericsson approved developer certificate is slightly different than otherwise, and is described below.

The image below gives an overview of the steps to go through for applications requiring basic and extended set capabilities compared to when also manufacturer approved capabilites are required.



The process for requesting a developer certificate not requiring phone manufacturer approval is briefly as follows:

1. Login to your user account on www.symbiansigned.com.

2. Download, install and run the DevCertRequest tool to generate a Certified Symbian Request (.csr) file in your computer. When running the tool, the following information is entered into its dialogs (more details on the tool can be found in the user guide included with the downloaded package):

- Name and path for the certificate request (.csr) file to be generated. It is possible to select an existing file that will be overwritten.
- Private key and password. An ACS publisher ID certificate and its private key can be used, in which case the tool allows more capabilities and IMEIs to be selected in the following steps. It is also possible to select an existing private key or let the tool generate a new one.
- Personal information. When an ACS publisher ID was selected, this information is automatically retrieved and can not be altered.
- Capabilites required and IMEIs of phones to be used for testing. Capabilities are selected from a list. IMEIs can be entered manually or imported from a text file.

When all information has been entered, the .csr and its private key (.key) files to be uploaded to the Symbian Signed portal are generated.

3. Request a developer certificate by uploading the .csr to the Symbian Signed Portal.

4. Check that the developer certificate has been created and that its capabilities and IMEIs are the requested.

5. Download the certificate. Together with the private key that the DevCertRequest tool generated, it can be used to sign .SIS files for installation and test on phones with the specified IMEIs.

Requesting a phone manufacturer approved developer certificate

The process for requesting a developer certificate requiring phone manufacturer approval is briefly as follows:

1. Login to your user account on www.symbiansigned.com.

2. Download, install and run the DevCertRequest tool to generate a Certified Symbian Request (.csr) file in your computer. When running the tool, the following information is entered into its dialogs (more details on the tool can be found in the user guide included with the downloaded package):

   - Name and path for the certificate request (.csr) file to be generated. It is possible to select an existing file that will be overwritten.
   - An ACS publisher ID certificate to be used and the corresponding private key.
   - Personal information is automatically retrieved from the ACS publisher ID certificate and can not be altered.
   - Capabilites required and IMEIs of phones to be used for testing. Capabilities are selected from a list. IMEIs can be entered manually or imported from a text file.

   When all information has been entered, the .csr and its private key (.key) files to be uploaded to the Symbian Signed portal are generated.

3. From the “Request Devcert” page in the Symbian Signed portal, click the link “Request Phone Manufacturer Approved DevCert”. This will start a workflow that is slightly different from the one where approval from Sony Ericsson is not required. After selecting Sony Ericsson as phone manufacturer, the following information has to be entered:

   - Information about the developer certificate
   - Developer contact details
   - Request details, including the number of IMEIs, rational for accessing the requested phone manufacturer capabilities and application details.
   - Business reasons for requiring access to restricted APIs

4. Upload the .csr file. When uploaded, the request is forwarded to Sony Ericsson, where a steering group reviews the request. Their decision is communicated to the developer via Symbian Signed.

5. When approved, the developer certificate can be viewed and checked and that its capabilities and IMEIs are the requested via login to the Symbian Signed account.

6. Download the certificate. Together with the private key that the DevCertRequest tool generated, it can be used to sign .SIS files for installation and test on phones with the specified IMEIs.

Note: Applications requiring capabilities from the phone manufacturer approved set, also have to go through a special process, involving Sony Ericsson approval, to be Symbiand Signed. This process is called *channel certification* and is described below, see “Submitting an application to Symbian Signed” on page 25

For more information on the Sony Ericsson criteria for approving developer certificates and other Sony Ericsson specific information on developer certificates, please refer to the FAQ found on http://developer.sonyericsson.com/site/global/gotomarket/certification/symbiancer/dev_certs_faq/p_devcerts_faq.jsp.

Testing procedures and tools

When testing an application to be Symbian Signed, a number of tests similar to the ones that Symbian Signed test houses do, are recommended to make the final testing and signing as smooth as possible. For that purpose a set of test tools are available from Symbian. Information about testing criteria and available test tools can be found on https://www.symbiansigned.com/app/page/requirements.

Submitting an application to Symbian Signed

Before sending an application to Symbian Signed to be tested and signed, there are several actions that should be handled with care to avoid the costly and time consuming event that the application is not approved and has to be changed in some way before it can be resubmitted.

To do before submitting the application

The Symbian Signed specification contains a list of tests that will be applied to an application to make sure the application is of a high quality. The Symbian Signed Test Criteria document can be found on https://www.symbiansigned.com/app/page/requirements.

The following things should also be checked, to make testing/signing procedures run smoothly.

- **UID Information**

  Make sure that the application UID (and VID if present) is owned by the submitter and from the correct dedicated range: UID = 0x20000000 - 0x2FFFFFFF, VID = 0x70000000 - 0x7FFFFFFF or VID = 0.

  Note: The owner of each UID, SID, or VID must match the ACS Publisher ID Distinguished Name (DN field) otherwise the application will fail the Symbian Signed process.

- **Policy statement dialog**

  The application should launch a policy statement dialog at the very first application startup, The dialog should summarize the use of network access, local connectivity, multimedia recording, read/write user data, phone call and location.

- **PKG file format**

  Make sure that "Localized Vendor name" and "Unique Vendor name" are consistent with the valid ACS publisher ID and with UID/SID/VID values. If the names are completely different, the app will be rejected by Symbian Signed.

  Make sure that the hardware dependency device UID is bracketed with [ ] instead of { } or ( ). More information about the pkg file format can be found in the UIQ 3 SDK documentation.

- **Sign the .SIS file with a valid ACS publish ID**

- **Contents of the package to submit for signing**

  The package should contain the .SIS and .pkg files together with a document giving the test house information about all functions of the application and how to use them. This guide may be a simple readme file or a complete user guide. The files to submit are compressed into a Zip archive.

Testing and signing procedures for Symbian Signed

1. In the Symbian Signed portal, a form is completed by the developer and the package containing the .pkg, .SIS and user guide files is uploaded. Symbian Signed responds with a receipt.
2. The selected test house verifies that the .SIS file is signed with a valid ACS publisher ID, and sends a quote by email.
3. When the test house receives payment, testing starts.
4. If the application fails the test, the developer has to make the requried changes and resubmit the application at a cost.
5. Once all tests have been passed, the test house resigns the application with a certificate linking to the Symbian root certificate installed on the targetet phones, sends an email to the developer and uploads the signed application to the developer account on the Symbian Signed portal.
6. The developer can login to the Symbian Signed account and download the signed application.

The Sony Ericsson channel certification path

As mentioned above, an application needing approval from Sony Ericsson to be signed, has to go through an extended signing process to be Symbian Signed. Here is an overview of the process:

1. The developer enters the required information into the Sony Ericsson Channel Certification application form, and submits it.
2. Sony Ericsson checks and reviews the submitted information against the previously made developer certificate request.

3. Sony Ericsson grants the developer access to a login protected Sony Ericsson channel certifier web page on the Symbian Signed website. Via this page the developer can submit the application for testing against the Symbian Signed criteria as well as the Sony Ericsson specific criteria.

4. The testing and signing process from now on follows the standard process as described above. The selected test house is responsible for testing and the final approval for signing of the application.

Appendix

Functions listed by capability

In the list, text within curly brackets { } indicates other capabilities that the function is associated with. The text “Dependent” in curly brackets indicates that the association with the capability is conditional.

Capability: AllFiles

CFileMan::Copy(const TDesC &,const TDesC &,TUint);{Dependent}

CFileMan::Copy(const TDesC &,const TDesC &,TUint,TRequestStatus &);{Dependent}

CFileMan::Move(const TDesC &,const TDesC &,TUint);{Dependent}

CFileMan::Move(const TDesC &,const TDesC &,TUint,TRequestStatus &);{Dependent}

MessageServer::CurrentDriveL(RFs &);{None}

MessageServer::DriveContainsStore(RFs &,TInt);{None}

RDir::Open(RFs &,const TDesC &,TUint);{}

RDir::Open(RFs &,const TDesC &,const TUidType &);{}

RFs::Att(const TDesC &,TUint &)const;{}

RFs::Modified(const TDesC &,TTime &)const;{}

RFs::NotifyChange(TNotifyType,TRequestStatus &);{}

RFs::NotifyChange(TNotifyType,TRequestStatus &,const TDesC &);{}

RFs::NotifyChangeCancel();{}

RFs::NotifyChangeCancel(TRequestStatus &);{}

Capability: CommDD

RBusDevComm::Open(TInt);{}

RBusDevCommDCE::Open(TInt);{}

RCommServ::CreateThreadInCommProc(const TDesC &,const TDesC &,TThreadFunction,TInt,TInt,TInt);{}

Capability: DiskAdmin

RFormat::Next(TInt &);{}

RFormat::Next(TPckgBuf< TInt > &,TRequestStatus &);{}

RFs::AddFileSystem(const TDesC &)const;{}

RFs::AddPlugin(const TDesC &)const;{}

RFs::CheckDisk(const TDesC &)const;{}

RFs::ClearPassword(TInt,const TMediaPassword &);{}

RFs::DismountFileSystem(const TDesC &,TInt)const;{}

RFs::DismountPlugin(const TDesC &)const;{}

RFs::DismountPlugin(const TDesC &,TInt)const;{}

RFs::DismountPlugin(const TDesC &,TInt,TInt)const;{}

RFs::ErasePassword(TInt);{}

RFs::FinaliseDrives();{}

RFs::LockDrive(TInt,const TMediaPassword &,const TMediaPassword &,TBool);{}

RFs::MountFileSystem(const TDesC &,TInt)const;{}

RFs::MountFileSystem(const TDesC &,TInt,TBool)const;{}

RFs::MountFileSystem(const TDesC &,const TDesC &,TInt);{}

RFs::MountFileSystem(const TDesC &,const TDesC &,TInt,TBool);{}

RFs::MountFileSystemAndScan(const TDesC &,TInt,TBool &)const;{}

RFs::MountPlugin(const TDesC &)const;{}

RFs::MountPlugin(const TDesC &,TInt)const;{}

RFs::MountPlugin(const TDesC &,TInt,TInt)const;{}

RFs::RemoveFileSystem(const TDesC &)const;{}

RFs::RemovePlugin(const TDesC &)const;{}

RFs::ScanDrive(const TDesC &)const;{}

RFs::SetDriveName(TInt,const TDesC &);{}

RFs::SetVolumeLabel(const TDesC &,TInt);{}

RFs::UnlockDrive(TInt,const TMediaPassword &,TBool);{}

RRawDisk::Open(RFs &,TInt);{}

RRawDisk::Read(TInt64,TDes8 &);{}

RRawDisk::Write(TInt64,TDesC8 &);{}

Capability: Drm

ContentAccess::CAgentContent::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &)=0;{}

ContentAccess::CAgentContent::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &,TRequestStatus &)=0;{}

ContentAccess::CAgentContent::CancelNotifyStatusChange(TRequestStatus &,const TDesC &)=0;{}

ContentAccess::CAgentContent::CancelRequestRights(TRequestStatus &,const TDesC &)=0;{}

ContentAccess::CAgentContent::CloseContainer()=0;{}

ContentAccess::CAgentContent::GetAttribute(TInt,TInt &,const TDesC &)=0;{}

ContentAccess::CAgentContent::GetAttributeSet(RAttributeSet &,const TDesC &)=0;{}

ContentAccess::CAgentContent::GetEmbeddedObjectsL(RStreamablePtrArray< CEmbeddedObject > &)=0;{}

ContentAccess::CAgentContent::GetEmbeddedObjectsL(RStreamablePtrArray< CEmbeddedObject > &,TEmbeddedType)=0;{}

ContentAccess::CAgentContent::GetStringAttribute(TInt,TDes &,const TDesC &)=0;{}

ContentAccess::CAgentContent::GetStringAttributeSet(RStringAttributeSet &,const TDesC &)=0;{}

ContentAccess::CAgentContent::NotifyStatusChange(TEventMask,TRequestStatus &,const TDesC &)=0;{}

ContentAccess::CAgentContent::OpenContainer(const TDesC &)=0;{}

ContentAccess::CAgentContent::Search(RStreamablePtrArray< CEmbeddedObject > &,const TDesC8 &,TBool)=0;{}

ContentAccess::CAgentContent::SetProperty(TAgentProperty,TInt)=0;{}

ContentAccess::CAgentData::DataSizeL(TInt &)=0;{}

ContentAccess::CAgentData::EvaluateIntent(TIntent)=0;{}

ContentAccess::CAgentData::ExecuteIntent(TIntent)=0;{}

ContentAccess::CAgentData::GetAttribute(TInt,TInt &)=0;{}

ContentAccess::CAgentData::GetStringAttribute(TInt,TDes &)=0;{}

ContentAccess::CAgentData::Read(TDes8 &)=0;{}

ContentAccess::CAgentData::Read(TDes8 &,TInt)=0;{}

ContentAccess::CAgentData::Read(TDes8 &,TInt,TRequestStatus &)=0;{}

ContentAccess::CAgentData::Read(TDes8 &,TRequestStatus &)=0;{}

ContentAccess::CAgentData::Seek(TSeek,TInt &)=0;{}

ContentAccess::CAgentData::SetProperty(TAgentProperty,TInt)=0;{}

ContentAccess::CAgentImportFile::GetImportStatus()const=0;{}

ContentAccess::CAgentImportFile::OutputFileL(TInt)=0;{}

ContentAccess::CAgentImportFile::WriteData(const TDesC8 &)=0;{}

ContentAccess::CAgentImportFile::WriteData(const TDesC8 &,TRequestStatus &)=0;{}

ContentAccess::CAgentImportFile::WriteDataComplete()=0;{}

ContentAccess::CAgentImportFile::WriteDataComplete(TRequestStatus &)=0;{}

ContentAccess::CAgentManager::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &)=0;{}

ContentAccess::CAgentManager::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &,TRequestStatus &)=0;{}

ContentAccess::CAgentManager::CancelNotifyStatusChange(const TDesC &,TRequestStatus &)=0;{}

ContentAccess::CAgentManager::CopyFile(const TDesC &,const TDesC &)=0;{}

ContentAccess::CAgentManager::DeleteFile(const TDesC &)=0;{}

ContentAccess::CAgentManager::GetDir(const TDesC &,TUint,TUint,CDir *&)const=0;{}

ContentAccess::CAgentManager::GetDir(const TDesC &,TUint,TUint,CDir *&,CDir *&)const=0;{}

ContentAccess::CAgentManager::GetDir(const TDesC &,const TUidType &,TUint,CDir *&)const=0;{}

ContentAccess::CAgentManager::MkDir(const TDesC &)=0;{}

ContentAccess::CAgentManager::MkDirAll(const TDesC &)=0;{}

ContentAccess::CAgentManager::NotifyStatusChange(const TDesC &,TEventMask,TRequestStatus &)=0;{}

ContentAccess::CAgentManager::RenameDir(const TDesC &,const TDesC &);{}

ContentAccess::CAgentManager::RenameFile(const TDesC &,const TDesC &)=0;{}

ContentAccess::CAgentManager::RmDir(const TDesC &)=0;{}

ContentAccess::CAgentManager::SetProperty(TAgentProperty,TInt)=0;{}

ContentAccess::CAgentRightsManager::DeleteAllRightsObjects(const TVirtualPathPtr &)=0;{}

ContentAccess::CAgentRightsManager::DeleteRightsObject(const CRightsInfo &)=0;{}

ContentAccess::CAgentRightsManager::GetRightsDataL(const CRightsInfo &)const=0;{}

ContentAccess::CAgentRightsManager::ListAllRightsL(RStreamablePtrArray< CRightsInfo > &)const=0;{}

ContentAccess::CAgentRightsManager::ListContentL(RStreamablePtrArray< CVirtualPath > &,CRightsInfo &)const=0;{}

ContentAccess::CAgentRightsManager::ListRightsL(RStreamablePtrArray< CRightsInfo > &,TVirtualPathPtr &)const=0;{}

ContentAccess::CAgentRightsManager::ListRightsL(RStreamablePtrArray< CRightsInfo > &,const TDesC &)const=0;{}

ContentAccess::CAgentRightsManager::SetProperty(TAgentProperty,TInt)=0;{}

ContentAccess::CContent::Agent()const;{}

ContentAccess::CContent::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &);{}

ContentAccess::CContent::AgentSpecificCommand(TInt,const TDesC8 &,TDes8 &,TRequestStatus &);{}

ContentAccess::CContent::CContent::OpenContentL(TIntent,TContentShareMode);{}

ContentAccess::CContent::CancelNotifyStatusChange(TRequestStatus &);{}

ContentAccess::CContent::CancelNotifyStatusChange(TRequestStatus &,const TDesC &);{}

ContentAccess::CContent::CancelRequestRights(TRequestStatus &);{}

ContentAccess::CContent::CancelRequestRights(TRequestStatus &,const TDesC &);{}

ContentAccess::CContent::CloseContainer();{}

ContentAccess::CContent::GetAttribute(TInt,TInt &)const;{}

ContentAccess::CContent::GetAttribute(TInt,TInt &,const TDesC &)const;{}

ContentAccess::CContent::GetAttributeSet(RAttributeSet &)const;{}

ContentAccess::CContent::GetAttributeSet(RAttributeSet &,const TDesC &)const;{}

ContentAccess::CContent::GetEmbeddedObjectsL(RStreamablePtrArray< CEmbeddedObject > &)const;{}

ContentAccess::CContent::GetEmbeddedObjectsL(RStreamablePtrArray< CEmbeddedObject > &,TEmbeddedType)const;{}

ContentAccess::CContent::GetStringAttribute(TInt,TDes &)const;{}

ContentAccess::CContent::GetStringAttribute(TInt,TDes &,const TDesC &)const;{}

ContentAccess::CContent::GetStringAttributeSet(RStringAttributeSet &)const;{}

ContentAccess::CContent::GetStringAttributeSet(RStringAttributeSet &,const TDesC &)const;{}

ContentAccess::CContent::NewAttributeL(TBool);{}

ContentAccess::CContent::NewAttributeL(TBool,TContentShareMode);{}

ContentAccess::CContent::NewL(RFile &);{}

ContentAccess::CContent::NewL(const TDesC &);{}

ContentAccess::CContent::NewL(const TDesC &,TContentShareMode);{}

ContentAccess::CContent::NewLC(RFile &);{}

ContentAccess::CContent::NewLC(const TDesC &);{}

ContentAccess::CContent::NewLC(const TDesC &,TContentShareMode);{}

ContentAccess::CContent::NotifyStatusChange(TEventMask,TRequestStatus &);{}

ContentAccess::CContent::NotifyStatusChange(TEventMask,TRequestStatus &,const TDesC &);{}

ContentAccess::CContent::OpenContainer(const TDesC &);{}

ContentAccess::CContent::OpenContentL(TIntent);{}

ContentAccess::CContent::OpenContentL(TIntent,const TDesC &);{}

ContentAccess::CContent::OpenContentLC(TIntent);{}

ContentAccess::CContent::OpenContentLC(TIntent,const TDesC &);{}

ContentAccess::CContent::Search(RStreamablePtrArray< CEmbeddedObject > &,const TDesC8 &,TBool);{}

ContentAccess::CContent::SetProperty(TAgentProperty,TInt);{}

ContentAccess::CData::DataSizeL(TInt &);{}

ContentAccess::CData::EvaluateIntent(TIntent);{}

ContentAccess::CData::ExecuteIntent(TIntent);{}

ContentAccess::CData::GetAttribute(TInt,TInt &)const;{}

ContentAccess::CData::GetAttributeSet(RAttributeSet &)const;{}

ContentAccess::CData::GetStringAttribute(TInt,TDes &)const;{}

ContentAccess::CData::GetStringAttributeSet(RStringAttributeSet &)const;{}

ContentAccess::CData::Read(TDes8 &)const;{}

ContentAccess::CData::Read(TDes8 &,TInt)const;{}

ContentAccess::CData::Read(TDes8 &,TInt,TRequestStatus &)const;{}

ContentAccess::CData::Read(TDes8 &,TRequestStatus &)const;{}

ContentAccess::CData::Seek(TSeek,TInt &)const;{}

ContentAccess::CData::SetProperty(TAgentProperty,TInt);{}

ContentAccess::CImportFile::GetImportStatus()const;{}

ContentAccess::CImportFile::OutputFileL(TInt)const;{}

ContentAccess::CImportFile::WriteData(const TDesC8 &);{}

ContentAccess::CImportFile::WriteData(const TDesC8 &,TRequestStatus &);{}

ContentAccess::CImportFile::WriteDataComplete();{}

ContentAccess::CImportFile::WriteDataComplete(TRequestStatus &);{}

ContentAccess::CManager::AgentSpecificCommand(TAgent &,TInt,const TDesC8 &,TDes8 &);{}

ContentAccess::CManager::AgentSpecificCommand(TAgent &,TInt,const TDesC8 &,TDes8 &,TRequest-Status &);{}

ContentAccess::CManager::CancelNotifyStatusChange(const TDesC &,TRequestStatus &);{}

ContentAccess::CManager::CopyFile(const TDesC &,const TDesC &)const;{}

ContentAccess::CManager::CreateRightsManagerL(TAgent &)const;{}

ContentAccess::CManager::DeleteFile(const TDesC &)const;{}

ContentAccess::CManager::DisplayManagementInfoL(TAgent &);{}

ContentAccess::CManager::GetDir(const TDesC &,TUint,TUint,CDir *&)const;{}

ContentAccess::CManager::GetDir(const TDesC &,TUint,TUint,CDir *&,CDir *&)const;{}

ContentAccess::CManager::GetDir(const TDesC &,const TUidType &,TUint,CDir *&)const;{}

ContentAccess::CManager::MkDir(const TDesC &)const;{}

ContentAccess::CManager::MkDirAll(const TDesC &)const;{}

ContentAccess::CManager::NotifyStatusChange(const TDesC &,TEventMask,TRequestStatus &);{}

ContentAccess::CManager::RenameFile(const TDesC &,const TDesC &)const;{}

ContentAccess::CManager::RmDir(const TDesC &)const;{}

ContentAccess::CManager::SetProperty(TAgentProperty,TInt);{}

ContentAccess::CRightsInfo::Description()const;{}

ContentAccess::CRightsInfo::RightsStatus()const;{}

ContentAccess::CRightsInfo::RightsType()const;{}

ContentAccess::CRightsInfo::UniqueId()const;{}

ContentAccess::CRightsManager::DeleteAllRightsObjects(const TVirtualPathPtr &);{}

ContentAccess::CRightsManager::DeleteRightsObject(const CRightsInfo &);{}

ContentAccess::CRightsManager::GetRightsDataL(const CRightsInfo &)const;{}

ContentAccess::CRightsManager::ListAllRightsL(RStreamablePtrArray< CRightsInfo > &)const;{}

ContentAccess::CRightsManager::ListContentL(RStreamablePtrArray< CVirtualPath > &,CRightsInfo &)const;{}

ContentAccess::CRightsManager::ListRightsL(RStreamablePtrArray< CRightsInfo > &,TVirtualPathPtr &)const;{}

ContentAccess::CRightsManager::ListRightsL(RStreamablePtrArray< CRightsInfo > &,const TDesC &)const;{}

ContentAccess::CRightsManager::SetProperty(TAgentProperty,TInt);{}

Capability: LocalServices

BAL::RBALClient::ConnectToHost();{NetworkServices}

BAL::RBALClient::DisconnectFromHostL();{NetworkServices}

BAL::RBALClient::SetConnectConfigurationL(const TBALConnectionConfig &);{NetworkServices}

BAL::RBALClient::StartListenerL();{NetworkServices}

BAL::RBALClient::StopListenerL();{NetworkServices}

CBluetoothSocket::NewL(MBluetoothSocketNotifier &,RSocketServ &,TUint,TUint);{}

CBluetoothSocket::NewL(MBluetoothSocketNotifier &,RSocketServ &,TUint,TUint,RConnection &);{}

CBluetoothSocket::NewL(MBluetoothSocketNotifier &,RSocketServ &,const TDesC &);{}

CBluetoothSocket::NewLC(MBluetoothSocketNotifier &,RSocketServ &,TUint,TUint);{}

CBluetoothSocket::NewLC(MBluetoothSocketNotifier &,RSocketServ &,TUint,TUint,RConnection &);{}

CBluetoothSocket::NewLC(MBluetoothSocketNotifier &,RSocketServ &,const TDesC &);{}

CBluetoothSynchronousLink::NewL(MBluetoothSynchronousLinkNotifier &,RSocketServ &);{}

CBluetoothSynchronousLink::NewLC(MBluetoothSynchronousLinkNotifier &,RSocketServ &);{}

CSdpAgent::AttributeRequestL(MSdpElementBuilder *,TSdpServRecordHandle,TSdpAttributeID);{}

CSdpAgent::AttributeRequestL(MSdpElementBuilder *,TSdpServRecordHandle,const CSdpAttrIdMatch-List &);{}

CSdpAgent::AttributeRequestL(TSdpServRecordHandle,TSdpAttributeID);{}

CSdpAgent::AttributeRequestL(TSdpServRecordHandle,const CSdpAttrIdMatchList &);{}

CSdpAgent::Cancel();{}

CSdpAgent::NewL(MSdpAgentNotifier &,const TBTDevAddr &);{}

CSdpAgent::NewLC(MSdpAgentNotifier &,const TBTDevAddr &);{}

CSdpAgent::NextRecordRequestL();{}

CSdpAgent::SetAttributePredictorListL(const CSdpAttrIdMatchList &);{}

CSdpAgent::SetRecordFilterL(const CSdpSearchPattern &);{}

RBTLocalDevice::Modify(const TBTLocalDevice &);{WriteDeviceData}

RBTLocalDevice::Modify(const TBTLocalDevice &,TRequestStatus &);{WriteDeviceData}

RBTPhysicalLinkAdapter::Open(RSocketServ &,RSocket &);{}

RBTPhysicalLinkAdapter::Open(RSocketServ &,TBTDevAddr &);{}

RBTRegServ::Connect();{}

RBTRegistry::CreateView(const TBTRegistrySearch &,TRequestStatus &);{ReadDeviceData}

RBTRegistry::DeleteAllInView(TRequestStatus &);{WriteDeviceData}

RBTRegistry::GetDevice(TBTNamelessDevice &,TRequestStatus &);{ReadDeviceData}

RBTRegistry::ModifyBluetoothDeviceNameL(const TBTDevAddr &,const TDesC8 &,TRequestStatus &);{WriteDeviceData}

RBTRegistry::ModifyDevice(const TBTNamelessDevice &,TRequestStatus &);{WriteDeviceData}

RBTRegistry::ModifyFriendlyDeviceNameL(const TBTDevAddr &,const TDesC &,TRequestStatus &);{}

RBTRegistry::UnpairAllInView(TRequestStatus &);{WriteDeviceData}

RBTRegistry::UnpairDevice(const TBTDevAddr &,TRequestStatus &);{WriteDeviceData}

RHCIDirectAccess::Open(RSocketServ &);{}

RSdpDatabase::Close();{}

RSdpDatabase::CreateServiceRecordL(CSdpAttrValueDES &,TSdpServRecordHandle &);{}

RSdpDatabase::CreateServiceRecordL(const TUUID &,TSdpServRecordHandle &);{}

RSdpDatabase::DeleteAttributeL(TSdpServRecordHandle,TSdpAttributeID);{}

RSdpDatabase::DeleteRecordL(TSdpServRecordHandle);{}

RSdpDatabase::Open(RSdp &);{}

RSdpDatabase::RSdpDatabase();{}

RSdpDatabase::UpdateAttributeL(TSdpServRecordHandle,TSdpAttributeID,CSdpAttrValue &);{}

RSdpDatabase::UpdateAttributeL(TSdpServRecordHandle,TSdpAttributeID,TUint);{}

RSdpDatabase::UpdateAttributeL(TSdpServRecordHandle,TSdpAttributeID,const TDesC16 &);{}

RSdpDatabase::UpdateAttributeL(TSdpServRecordHandle,TSdpAttributeID,const TDesC8 &);{}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId);{NetworkServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlSyncType);{NetworkServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlSyncType,const RArray< TSmlTaskId > &);{NetworkServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,const RArray< TSmlTaskId > &);{NetworkServices}

RSyncMLDevManJob::CreateL(RSyncMLSession &,TSmlProfileId);{NetworkServices}

RSyncMLDevManJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlConnectionId);{NetworkServices}

RSyncMLTransport::StartListeningL()const;{}

RSyncMLTransport::StopListeningL()const;{}

Capability: Location

CTelephony::GetCurrentNetworkInfo(TRequestStatus &,TDes8 &)const;{ReadDeviceData, ReadUserData}

RMobilePhone::GetCurrentNetwork(TRequestStatus &,TDes8 &,TMobilePhoneLocationAreaV1 &)const;{ReadDeviceData}

RMobilePhone::NotifyCurrentNetworkChange(TRequestStatus &,TDes8 &,TMobilePhoneLocationAreaV1 &)const;{ReadDeviceData}

Capability: MultimediaDD

CCamera::NewL(MCameraObserver2 &,TInt,TInt);{UserEnvironment}

CDevASR::SetPrioritySettings(const TMMFPrioritySettings &);{}

CMMFController::SetPrioritySettings(const TMMFPrioritySettings &)=0;{}

CMMFUrlSink::SetSinkPrioritySettings(const TMMFPrioritySettings &);{}

CMMFUrlSource::SetSourcePrioritySettings(const TMMFPrioritySettings &);{}

CMMTunerUtility::SetPriority(TTunerAccessPriority);{}

CMdaAudioConvertUtility::NewL(MMdaObjectStateChangeObserver &,CMdaServer *,TInt,TMdaPriorityPreference);{}

CMdaAudioInputStream::NewL(MMdaAudioInputStreamCallback &,TInt,TMdaPriorityPreference);{UserEnvironment}

CMdaAudioInputStream::SetPriority(TInt,TMdaPriorityPreference);{}

CMdaAudioOutputStream::NewL(MMdaAudioOutputStreamCallback &,TInt,TMdaPriorityPreference);{}

CMdaAudioOutputStream::SetPriority(TInt,TMdaPriorityPreference);{}

CMdaAudioPlayerUtility::NewDesPlayerL(const TDesC8 &,MMdaAudioPlayerCallback &,TInt,TMdaPriorityPreference,CMdaServer *);{}

CMdaAudioPlayerUtility::NewFilePlayerL(const TDesC &,MMdaAudioPlayerCallback &,TInt,TMdaPriorityPreference,CMdaServer *);{}

CMdaAudioPlayerUtility::NewL(MMdaAudioPlayerCallback &,TInt,TMdaPriorityPreference);{}

CMdaAudioPlayerUtility::SetPriority(TInt,TMdaPriorityPreference);{}

CMdaAudioRecorderUtility::NewL(MMdaObjectStateChangeObserver &,CMdaServer *,TInt,TMdaPriorityPreference);{}

CMdaAudioRecorderUtility::SetPriority(TInt,TMdaPriorityPreference);{}

CMdaAudioToneUtility::NewL(MMdaAudioToneObserver &,CMdaServer *,TInt,TMdaPriorityPreference);{}

CMdaAudioToneUtility::SetPriority(TInt,TMdaPriorityPreference);{}

CMidiClientUtility::NewL(MMidiClientUtilityObserver &,TInt,TMdaPriorityPreference);{}

CMidiClientUtility::SetPriorityL(TInt,TMdaPriorityPreference);{}

CMmfGlobalAudioEffect::SetByValuesL(const TDesC8 &);{}

CMmfGlobalAudioEffect::SetEnabledL(TBool);{}

CMmfGlobalAudioEffect::SetSettingsByDesL(const TDesC8 &);{}

CMmfGlobalAudioEffect::SetSettingsByUidL(TUid);{}

CSpeechRecognitionUtility::SetAudioPriority(TInt,TInt,TInt,TInt);{}

CVideoPlayerUtility::NewL(MVideoPlayerUtilityObserver &,TInt,TMdaPriorityPreference,RWsSession &,CWsScreenDevice &,RWindowBase &,const TRect &,const TRect &);{}

CVideoPlayerUtility::SetPriorityL(TInt,TMdaPriorityPreference);{}

CVideoRecorderUtility::NewL(MVideoRecorderUtilityObserver &,TInt,TMdaPriorityPreference);{}

CVideoRecorderUtility::SetPriorityL(TInt,TMdaPriorityPreference);{}

MDataSink::SetSinkPrioritySettings(const TMMFPrioritySettings &);{}

MDataSource::SetSourcePrioritySettings(const TMMFPrioritySettings &);{}

MMMTunerUtilityImpl::SetPriority(CMMTunerUtility::TTunerAccessPriority)=0;{}

MMmfGlobalAudioImpl::SetByValuesL(const TDesC8 &)=0;{}

MMmfGlobalAudioImpl::SetEnabledL(TBool)=0;{}

MMmfGlobalAudioImpl::SetSettingsByDesL(const TDesC8 &)=0;{}

MMmfGlobalAudioImpl::SetSettingsByUidL(TUid)=0;{}

RMMFController::Open(TUid,const TMMFPrioritySettings &);{}

RMMFController::Open(const CMMFControllerImplementationInformation &,const TMMFPrioritySettings &);{}

RMMFController::SetPrioritySettings(const TMMFPrioritySettings &)const;{}

RMdaDevSound::Open(TInt);{}

Capability: NetworkControl

CBluetoothPhysicalLinks::Broadcast(const TDesC8 &);{}

CBluetoothPhysicalLinks::Disconnect(const TBTDevAddr &);{}

CBluetoothPhysicalLinks::DisconnectAll();{}

CBluetoothPhysicalLinks::ReadRaw(TDes8 &);{}

RCall::AnswerIncomingCall()const;{NetworkServices}

RCall::AnswerIncomingCall(TRequestStatus &);{NetworkServices}

RCall::AnswerIncomingCall(TRequestStatus &,const TDesC8 &);{NetworkServices}

RCall::AnswerIncomingCall(const TDesC8 &)const;{NetworkServices}

RCall::AnswerIncomingCallCancel()const;{NetworkServices}

RCall::Dial(TRequestStatus &,const TDesC8 &,const TTelNumberC &);{NetworkServices}

RCall::Dial(TRequestStatus &,const TTelNumberC &);{NetworkServices}

RCall::Dial(const TDesC8 &,const TTelNumberC &)const;{NetworkServices}

RCall::Dial(const TTelNumberC &)const;{NetworkServices}

RCall::DialCancel()const;{NetworkServices}

RCall::LoanDataPort(TCommPort &)const;{NetworkServices}

RCall::LoanDataPort(TRequestStatus &,TCommPort &);{NetworkServices}

RCall::RecoverDataPort()const;{NetworkServices}

RCdmaMobilePhone::EndEmergencyMode(TRequestStatus &)const;{NetworkServices}

RCdmaMobilePhone::SetCallProcessingSuspendState(TRequestStatus &,TBool)const;{NetworkServices}

RConnection::Control(TUint,TUint,TDes8 &);{Dependent}

RConnection::Ioctl(TUint,TUint,TRequestStatus &,TDes8 *);{Dependent}

RFax::Open(RCall &);{}

RHostResolver::SetHostName(const TDesC &);{}

RMobileCall::DialEmergencyCall(TRequestStatus &,const TDesC &)const;{NetworkServices}

RMobileCall::DialNoFdnCheck(TRequestStatus &,const TDesC &)const;{NetworkServices}

RMobileCall::DialNoFdnCheck(TRequestStatus &,const TDesC8 &,const TDesC &)const;{NetworkServices}

RMobileCall::GetMobileDataCallRLPRange(TRequestStatus &,TInt,TDes8 &)const;{ReadDeviceData}

RMobilePhone::GetDefaultPrivacy(TMobilePhonePrivacy &)const;{ReadDeviceData}

RMobilePhone::InitialiseMM(TRequestStatus &,TDes8 &)const;{WriteDeviceData}

RMobilePhone::NotifyDefaultPrivacyChange(TRequestStatus &,TMobilePhonePrivacy &)const;{ReadDeviceData}

RMobilePhone::SendNetworkServiceRequestNoFdnCheck(TRequestStatus &,const TDesC &)const;{NetworkServices, WriteDeviceData}

RMobilePhone::SetDefaultPrivacy(TRequestStatus &,TMobilePhonePrivacy)const;{WriteDeviceData}

RMobilePhone::SetPersonalisationStatus(TRequestStatus &,const TMobilePhonePersonalisation,const TDes &)const;{WriteDeviceData}

RMobilePhone::SetSmartCardApplicationStatus(TRequestStatus &,const TAID &,TSmartCardApplicationAction)const;{WriteDeviceData}

RMobilePhone::UpdateScFile(TRequestStatus &,const TScFilePathWithAccessOffsets &,TDes8 &)const;{WriteDeviceData}

RMobileSmsMessaging::AckSmsStored(TRequestStatus &,const TDesC8 &,TBool)const;{NetworkServices}

RMobileSmsMessaging::NackSmsStored(TRequestStatus &,const TDesC8 &,TInt)const;{NetworkServices}

RMobileSmsMessaging::ResumeSmsReception(TRequestStatus &)const;{NetworkServices}

RMobileSmsMessaging::SendMessageNoFdnCheck(TRequestStatus &,const TDesC8 &,TDes8 &)const;{NetworkServices}

RMobileUssdMessaging::ReceiveMessage(TRequestStatus &,TDes8 &,TDes8 &)const;{ReadDeviceData}

RMobileUssdMessaging::SendMessage(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkServices, WriteDeviceData}

RMobileUssdMessaging::SendMessageNoFdnCheck(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkServices, WriteDeviceData}

RMobileUssdMessaging::SendRelease(TRequestStatus &,TDes8 &)const;{NetworkServices}

RPhone::Initialise();{}

RPhone::Initialise(TRequestStatus &);{}

RPhone::InitialiseCancel();{}

RQoSPolicy::RQoSPolicy();{}

RQoSPolicy::~RQoSPolicy();{}

RRootServ::Bind(TRequestStatus &,TRSBindingInfo &);{}

RRootServ::LoadCpm(TRequestStatus &,const TRSStartModuleParams &,const TDesC8 &);{}

RRootServ::SetMBufPoolSize(TUint);{}

RRootServ::Shutdown();{}

RRootServ::Unbind(TRequestStatus &,TRSUnBindingInfo &);{}

RRootServ::UnloadCpm(TRequestStatus &,const TCFModuleName &,TRSUnLoadType);{}

RSADB::FinalizeAndSend(TPfkeySendMsgBase &,TRequestStatus &);{}

RSADB::ReadRequest(TDes8 &,TRequestStatus &);{}

RSADB::SendRequest(const TDesC8 &,TRequestStatus &);{}

RSat::ClientSatProfileIndication(const TDesC8 &)const;{WriteDeviceData}

RSat::EventDownload(TRequestStatus &,TEventList,const TDesC8 &)const;{WriteDeviceData}

RSat::MenuSelection(TRequestStatus &,const TDesC8 &)const;{ReadDeviceData}

RSat::NotifyCallControlRequest(TRequestStatus &,TDes8 &)const;{ReadDeviceData}

RSat::NotifyCbDownload(TRequestStatus &,TDes8 &)const;{ReadUserData}

RSat::NotifyMoSmControlRequest(TRequestStatus &,TDes8 &)const;{ReadDeviceData}

RSat::NotifySmsPpDownload(TRequestStatus &,TDes8 &)const;{ReadUserData}

RSat::TerminalRsp(TRequestStatus &,TPCmd,const TDesC8 &)const;{}

RSat::UsatClientReadyIndication()const;{}

RServiceResolver::RegisterService(const TDesC &,const TUint &);{}

RServiceResolver::RegisterService(const TDesC &,const TUint &,TRequestStatus &);{}

RServiceResolver::RemoveService(const TDesC &,const TUint &);{}

RServiceResolver::RemoveService(const TDesC &,const TUint &,TRequestStatus &);{}

RSocketServ::InstallExtension(const TDesC &,const TDesC &);{}

RSocketServ::SetExclusiveMode(TRequestStatus &);{}

RSocketServ::StartProtocol(TUint,TUint,TUint,TRequestStatus &);{}

RSocketServ::StopProtocol(TUint,TUint,TUint,TRequestStatus &);{}

RTelServer::SetPriorityClient()const;{}

Capability: NetworkServices

BAL::RBALClient::ConnectToHost();{LocalServices}

BAL::RBALClient::DisconnectFromHostL();{LocalServices}

BAL::RBALClient::SetConnectConfigurationL(const TBALConnectionConfig &);{LocalServices}

BAL::RBALClient::StartListenerL();{LocalServices}

BAL::RBALClient::StopListenerL();{LocalServices}

CAsyncRetrievePhoneList::CancelReq(TInt,TInt);{ReadDeviceData}

CAsyncRetrieveVariableLengthBuffer::DoCancel();{ReadDeviceData}

CExtensionBase::CExtensionBase();{}

CExtensionBase::Copy(const CExtensionBase &)=0;{}

CExtensionBase::CreateL()=0;{}

CExtensionBase::Data()=0;{}

CExtensionBase::ParseMessage(const TDesC8 &)=0;{}

CExtensionBase::Type()const;{}

CExtensionBase::~CExtensionBase();{}

CFaxTransfer::Start(TRequestStatus &);{ReadUserData, WriteUserData}

CFaxTransfer::Stop();{ReadUserData, WriteUserData}

CRetrieveMobilePhoneCBList::Start(TRequestStatus &,RMobilePhone::TMobilePhoneCBCondition,RMobilePhone::TMobileInfoLocation);{ReadDeviceData}

CRetrieveMobilePhoneCFList::Start(TRequestStatus &,RMobilePhone::TMobilePhoneCFCondition,RMobilePhone::TMobileInfoLocation);{ReadDeviceData}

CRetrieveMobilePhoneCWList::Start(TRequestStatus &,RMobilePhone::TMobileInfoLocation);{ReadDeviceData}

CRetrieveMobilePhoneCcbsList::Start(TRequestStatus &);{ReadDeviceData}

CRetrieveMobilePhoneDetectedNetworks::Start(TRequestStatus &);{ReadDeviceData}

CRetrieveMobilePhoneDetectedNetworks::StartV2(TRequestStatus &);{ReadDeviceData}

CSblpPolicy::Copy(const CExtensionBase &);{}

CSblpPolicy::CreateL();{}

CSblpPolicy::Data();{}

CSblpPolicy::ParseMessage(const TDesC8 &);{}

CTelephony::AnswerIncomingCall(TRequestStatus &,TCallId &,const TPhoneLine)const;{}

CTelephony::DialNewCall(TRequestStatus &,TDes8 &,const TTelNumber &,TCallId &,const TPhoneLine)const;{}

CTelephony::GetCallBarringStatus(TRequestStatus &,const TCallBarringCondition,TDes8 &,const TServiceGroup)const;{ReadDeviceData}

CTelephony::GetCallForwardingStatus(TRequestStatus &,const TCallForwardingCondition,TDes8 &,const TServiceGroup)const;{ReadDeviceData}

CTelephony::GetCallWaitingStatus(TRequestStatus &,TDes8 &,const TServiceGroup)const;{ReadDeviceData}

CTelephony::Hangup(TRequestStatus &,const TCallId &)const;{}

CTelephony::Hold(TRequestStatus &,const TCallId &)const;{}

CTelephony::Resume(TRequestStatus &,const TCallId &)const;{}

CTelephony::SendDTMFTones(TRequestStatus &,const TDesC &)const;{}

CTelephony::Swap(TRequestStatus &,const TCallId &,const TCallId &)const;{}

MQoSObserver::Event(const CQoSEventBase &)=0;{}

RCall::AcquireOwnership(TRequestStatus &)const;{}

RCall::AcquireOwnershipCancel()const;{}

RCall::AdoptFaxSharedHeaderFile(const RFile &)const;{}

RCall::AnswerIncomingCall()const;{NetworkControl}

RCall::AnswerIncomingCall(TRequestStatus &);{NetworkControl}

RCall::AnswerIncomingCall(TRequestStatus &,const TDesC8 &);{NetworkControl}

RCall::AnswerIncomingCall(const TDesC8 &)const;{NetworkControl}

RCall::AnswerIncomingCallCancel()const;{NetworkControl}

RCall::Connect()const;{}

RCall::Connect(TRequestStatus &);{}

RCall::Connect(TRequestStatus &,const TDesC8 &);{}

RCall::Connect(const TDesC8 &)const;{}

RCall::ConnectCancel()const;{}

RCall::Dial(TRequestStatus &,const TDesC8 &,const TTelNumberC &);{NetworkControl}

RCall::Dial(TRequestStatus &,const TTelNumberC &);{NetworkControl}

RCall::Dial(const TDesC8 &,const TTelNumberC &)const;{NetworkControl}

RCall::Dial(const TTelNumberC &)const;{NetworkControl}

RCall::DialCancel()const;{NetworkControl}

RCall::HangUp()const;{}

RCall::HangUp(TRequestStatus &)const;{}

RCall::HangUpCancel()const;{}

RCall::LoanDataPort(TCommPort &)const;{NetworkControl}

RCall::LoanDataPort(TRequestStatus &,TCommPort &);{NetworkControl}

RCall::RecoverDataPort()const;{NetworkControl}

RCall::TransferOwnership()const;{}

RCdmaMobileCall::NotifyIncomingNetworkFlashWithInfo(TRequestStatus &,TDes8 &)const;{ReadUserData}

RCdmaMobileCall::ResumeConnect(TRequestStatus &,const TBool)const;{}

RCdmaMobileCall::SendNetworkFlashWithInfo(TRequestStatus &,const TDes8 &)const;{}

RCdmaMobilePhone::EndEmergencyMode(TRequestStatus &)const;{NetworkControl}

RCdmaMobilePhone::ProcessOtaRequest(TRequestStatus &,const TDesC8 &,TDes8 &)const;{ReadDeviceData, WriteDeviceData}

RCdmaMobilePhone::SetCallProcessingSuspendState(TRequestStatus &,TBool)const;{NetworkControl}

RCdmaMobilePhone::StartOta(TRequestStatus &,TOtaServiceType,const TDes8 &)const;{WriteDeviceData}

RCdmaMobilePhone::StopOta(TRequestStatus &,TOtaServiceType)const;{WriteDeviceData}

RFax::Read(TRequestStatus &,TDes8 &);{ReadUserData}

RFax::TerminateFaxSession()const;{}

RFax::Write(TRequestStatus &,const TDesC8 &);{WriteUserData}

RLine::NotifyIncomingCall(TRequestStatus &,TName &);{}

RLine::NotifyIncomingCallCancel()const;{}

RMobileCall::ActivateCCBS(TRequestStatus &,TInt &)const;{}

RMobileCall::ActivateUUS(TRequestStatus &,const TDesC8 &)const;{WriteDeviceData}

RMobileCall::AnswerIncomingCallISV(TRequestStatus &,const TDesC8 &);{}

RMobileCall::AnswerIncomingCallWithUUI(TRequestStatus &,const TDesC8 &,const TMobileCallUUI &)const;{WriteUserData}

RMobileCall::AnswerMultimediaCallAsVoice(TRequestStatus &,const TDesC8 &,TName &)const;{}

RMobileCall::Deflect(TRequestStatus &,TMobileCallDeflect,const RMobilePhone::TMobileAddress &)const;{}

RMobileCall::DialEmergencyCall(TRequestStatus &,const TDesC &)const;{NetworkControl}

RMobileCall::DialISV(TRequestStatus &,const TDesC8 &,const TTelNumberC &);{}

RMobileCall::DialNoFdnCheck(TRequestStatus &,const TDesC &)const;{NetworkControl}

RMobileCall::DialNoFdnCheck(TRequestStatus &,const TDesC8 &,const TDesC &)const;{NetworkControl}

RMobileCall::GoOneToOne(TRequestStatus &)const;{}

RMobileCall::HangupWithUUI(TRequestStatus &,const TMobileCallUUI &)const;{WriteUserData}

RMobileCall::Hold(TRequestStatus &)const;{}

RMobileCall::NotifyPrivacyConfirmation(TRequestStatus &,RMobilePhone::TMobilePhonePrivacy &)const;{}

RMobileCall::NotifyTrafficChannelConfirmation(TRequestStatus &,TMobileCallTch &)const;{}

RMobileCall::ReceiveUUI(TRequestStatus &,TMobileCallUUI &)const;{ReadUserData}

RMobileCall::RejectCCBS()const;{}

RMobileCall::Resume(TRequestStatus &)const;{}

RMobileCall::SendUUI(TRequestStatus &,TBool,const TMobileCallUUI &)const;{WriteUserData}

RMobileCall::SetDynamicHscsdParams(TRequestStatus &,TMobileCallAiur,TInt)const;{}

RMobileCall::SetPrivacy(RMobilePhone::TMobilePhonePrivacy)const;{}

RMobileCall::SetTrafficChannel(TMobileCallTch)const;{}

RMobileCall::Swap(TRequestStatus &)const;{}

RMobileCall::SwitchAlternatingCall(TRequestStatus &)const;{}

RMobileCall::Transfer(TRequestStatus &)const;{}

RMobileConferenceCall::AddCall(TRequestStatus &,const TName &)const;{}

RMobileConferenceCall::CreateConference(TRequestStatus &)const;{}

RMobileConferenceCall::HangUp(TRequestStatus &)const;{}

RMobileConferenceCall::Swap(TRequestStatus &)const;{}

RMobilePhone::AcceptCCBSRecall(TRequestStatus &,TInt,TName &)const;{}

RMobilePhone::ContinueDTMFStringSending(TBool)const;{}

RMobilePhone::DeactivateCCBS(TRequestStatus &,TInt)const;{WriteDeviceData}

RMobilePhone::NotifyStopInDTMFString(TRequestStatus &)const;{}

RMobilePhone::ReadDTMFTones(TRequestStatus &,TDes &)const;{}

RMobilePhone::RefuseCCBSRecall(TInt)const;{}

RMobilePhone::SelectNetwork(TRequestStatus &,TBool,const TMobilePhoneNetworkManualSelection &)const;{WriteDeviceData}

RMobilePhone::SendDTMFTones(TRequestStatus &,const TDesC &)const;{}

RMobilePhone::SendNetworkServiceRequest(TRequestStatus &,const TDesC &)const;{WriteDeviceData}

RMobilePhone::SendNetworkServiceRequestNoFdnCheck(TRequestStatus &,const TDesC &)const;{NetworkControl, WriteDeviceData}

RMobilePhone::SetCallBarringPassword(TRequestStatus &,const TMobilePhonePasswordChangeV1 &)const;{WriteDeviceData}

RMobilePhone::SetCallBarringStatus(TRequestStatus &,TMobilePhoneCBCondition,const TMobilePhoneCBChangeV1 &)const;{WriteDeviceData}

RMobilePhone::SetCallForwardingStatus(TRequestStatus &,TMobilePhoneCFCondition,const TMobilePhoneCFChangeV1 &)const;{WriteDeviceData}

RMobilePhone::SetCallWaitingStatus(TRequestStatus &,TMobileService,TMobilePhoneServiceAction)const;{WriteDeviceData}

RMobilePhone::SetSSPassword(TRequestStatus &,const TDesC8 &,const TInt)const;{WriteDeviceData}

RMobilePhone::SetUUSSetting(TRequestStatus &,TMobilePhoneUUSSetting)const;{WriteDeviceData}

RMobilePhone::StartDTMFTone(TChar)const;{}

RMobilePhone::StopDTMFTone()const;{}

RMobilePhone::TerminateAllCalls(TRequestStatus &)const;{}

RMobileSmsMessaging::AckSmsStored(TRequestStatus &,const TDesC8 &,TBool)const;{NetworkControl}

RMobileSmsMessaging::NackSmsStored(TRequestStatus &,const TDesC8 &,TInt)const;{NetworkControl}

RMobileSmsMessaging::ReceiveMessage(TRequestStatus &,TDes8 &,TDes8 &)const;{ReadUserData}

RMobileSmsMessaging::ResumeSmsReception(TRequestStatus &)const;{NetworkControl}

RMobileSmsMessaging::SendMessage(TRequestStatus &,const TDesC8 &,TDes8 &)const;{WriteUserData}

RMobileSmsMessaging::SendMessageNoFdnCheck(TRequestStatus &,const TDesC8 &,TDes8 &)const;{NetworkControl}

RMobileUssdMessaging::SendMessage(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkControl, WriteDeviceData}

RMobileUssdMessaging::SendMessageNoFdnCheck(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkControl, WriteDeviceData}

RMobileUssdMessaging::SendRelease(TRequestStatus &,TDes8 &)const;{NetworkControl}

RPacketContext::Activate(TRequestStatus &)const;{}

RPacketContext::AddMediaAuthorizationL(TRequestStatus &,CTFTMediaAuthorizationV3 &)const;{WriteDeviceData}

RPacketContext::AddPacketFilter(TRequestStatus &,const TDesC8 &)const;{WriteDeviceData}

RPacketContext::CreateNewTFT(TRequestStatus &,const TInt)const;{}

RPacketContext::Deactivate(TRequestStatus &)const;{}

RPacketContext::DeleteTFT(TRequestStatus &)const;{}

RPacketContext::GetPacketFilterInfo(TRequestStatus &,TInt,TDes8 &)const;{ReadDeviceData}

RPacketContext::InitialiseContext(TRequestStatus &,TDes8 &)const;{}

RPacketContext::LoanCommPort(TRequestStatus &,RCall::TCommPort &)const;{}

RPacketContext::ModifyActiveContext(TRequestStatus &)const;{}

RPacketContext::RecoverCommPort(TRequestStatus &)const;{}

RPacketContext::RemoveMediaAuthorization(TRequestStatus &,TAuthorizationToken &)const;{WriteDeviceData}

RPacketContext::RemovePacketFilter(TRequestStatus &,TInt)const;{WriteDeviceData}

RPacketQoS::SetProfileParameters(TRequestStatus &,TDes8 &)const;{WriteDeviceData}

RPacketService::Attach(TRequestStatus &)const;{}

RPacketService::DeactivateNIF(TRequestStatus &,const TDesC &)const;{}

RPacketService::Detach(TRequestStatus &)const;{}

RPacketService::RejectActivationRequest(TRequestStatus &)const;{}

RPacketService::SetMSClass(TRequestStatus &,TMSClass)const;{WriteDeviceData}

RPacketService::SetPreferredBearer(TRequestStatus &,TPreferredBearer)const;{WriteDeviceData}

RSat::SendMessageNoLogging(TRequestStatus &,const TDesC8 &,TUint16 &)const;{WriteDeviceData}

RSocket::Ioctl(TUint,TRequestStatus &,TDes8 *,TUint);{Dependent}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId);{LocalServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlSyncType);{LocalServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlSyncType,const RArray< TSmlTaskId > &);{LocalServices}

RSyncMLDataSyncJob::CreateL(RSyncMLSession &,TSmlProfileId,const RArray< TSmlTaskId > &);{LocalServices}

RSyncMLDevManJob::CreateL(RSyncMLSession &,TSmlProfileId);{LocalServices}

RSyncMLDevManJob::CreateL(RSyncMLSession &,TSmlProfileId,TSmlConnectionId);{LocalServices}

Capability: PowerMgmt

Power::CancelWakeupEventNotification();{}

Power::DisableWakeupEvents();{}

Power::EnableWakeupEvents(TPowerState);{}

Power::PowerDown();{}

Power::RequestWakeupEventNotification(TRequestStatus &);{}

RProcess::Kill(TInt);{}

RProcess::Panic(const TDesC &,TInt);{}

RProcess::Terminate(TInt);{}

RWsSession::RequestOffEvents(TBool,RWindowTreeNode *);{}

UserHal::SwitchOff();{}

Capability: ProtServ

CServer2::Start(const TDesC &);{}

CServer2::StartL(const TDesC &);{}

User::SetCritical(TCritical);{}

User::SetProcessCritical(TCritical);{}

Capability: ReadDeviceData

CAsyncRetrieveAuthorizationInfo::Start(TRequestStatus &);{}

CAsyncRetrievePhoneList::CancelReq(TInt,TInt);{NetworkServices}

CAsyncRetrieveVariableLengthBuffer::DoCancel();{NetworkServices}

CRetrieveMobilePhoneCBList::RetrieveListL();{}

CRetrieveMobilePhoneCBList::Start(TRequestStatus &,RMobilePhone::TMobilePhoneCBCondition,RMobilePhone::TMobileInfoLocation);{NetworkServices}

CRetrieveMobilePhoneCFList::Start(TRequestStatus &,RMobilePhone::TMobilePhoneCFCondition,RMobilePhone::TMobileInfoLocation);{NetworkServices}

CRetrieveMobilePhoneCWList::RetrieveListL();{}

CRetrieveMobilePhoneCWList::Start(TRequestStatus &,RMobilePhone::TMobileInfoLocation);{NetworkServices}

CRetrieveMobilePhoneCcbsList::RetrieveListL();{}

CRetrieveMobilePhoneCcbsList::Start(TRequestStatus &);{NetworkServices}

CRetrieveMobilePhoneDetectedNetworks::Start(TRequestStatus &);{NetworkServices}

CRetrieveMobilePhoneDetectedNetworks::StartV2(TRequestStatus &);{NetworkServices}

CRetrieveMobilePhoneNamList::RetrieveListL();{}

CRetrieveMobilePhoneNamList::RetrieveListV4L();{}

CRetrieveMobilePhoneNamList::Start(TRequestStatus &,TInt);{}

CRetrieveMobilePhoneNamList::StartV4(TRequestStatus &,TInt);{}

CRetrieveMobilePhonePreferredNetworks::Start(TRequestStatus &);{}

CRetrieveMobilePhoneSmspList::RetrieveListL();{}

CRetrieveMobilePhoneSmspList::Start(TRequestStatus &);{}

CSmsSimParamOperation::DoReadSimParamsL();{}

CSmsSimParamOperation::DoRunReadSimParamsL();{}

CSmsSimParamOperation::ReadSimParamsL(TUid,TMsvId,CMsvSession &,TRequestStatus &);{}

CSmsSimParamOperation::RestoreSimParamsL(CMsvStore &,CMobilePhoneSmspList &);{}

CSmsSimParamOperation::TransferCommandL(TInt);{WriteDeviceData}

CTelephony::GetCallBarringStatus(TRequestStatus &,const TCallBarringCondition,TDes8 &,const TServiceGroup)const;{NetworkServices}

CTelephony::GetCallForwardingStatus(TRequestStatus &,const TCallForwardingCondition,TDes8 &,const TServiceGroup)const;{NetworkServices}

CTelephony::GetCallWaitingStatus(TRequestStatus &,TDes8 &,const TServiceGroup)const;{NetworkServices}

CTelephony::GetCurrentNetworkInfo(TRequestStatus &,TDes8 &)const;{Location, ReadUserData}

CTelephony::GetLockInfo(TRequestStatus &,const TIccLock &,TDes8 &)const;{}

CTelephony::GetPhoneId(TRequestStatus &,TDes8 &)const;{}

CTelephony::GetSubscriberId(TRequestStatus &,TDes8 &)const;{}

RBTRegistry::CreateView(const TBTRegistrySearch &,TRequestStatus &);{LocalServices}

RBTRegistry::GetDevice(TBTNamelessDevice &,TRequestStatus &);{LocalServices}

RCdmaMobilePhone::ProcessOtaRequest(TRequestStatus &,const TDesC8 &,TDes8 &)const;{Network-Services, WriteDeviceData}

RCdmaMobilePhone::ReadOtaStoreBlock(TRequestStatus &,TDes8 &,TDes8 &)const;{}

RMobileCall::GetMobileDataCallRLPRange(TRequestStatus &,TInt,TDes8 &)const;{NetworkControl}

RMobileConferenceCall::GetMobileCallInfo(TInt,TDes8 &)const;{}

RMobilePhone::EnumerateAPNEntries(TRequestStatus &,TUint32 &)const;{}

RMobilePhone::GetAPNname(TRequestStatus &,const TUint32,TDes8 &)const;{}

RMobilePhone::GetCurrentNetwork(TRequestStatus &,TDes8 &,TMobilePhoneLocationAreaV1 &)const;{Location}

RMobilePhone::GetCurrentNetworkName(TRequestStatus &,TDes8 &,TDes8 &)const;{}

RMobilePhone::GetDefaultPrivacy(TMobilePhonePrivacy &)const;{NetworkControl}

RMobilePhone::GetFeatureCode(TRequestStatus &,TDes &,TMobilePhoneNetworkService,TMobile-PhoneServiceAction)const;{}

RMobilePhone::GetHomeNetwork(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::GetIccIdentity(TRequestStatus &,TIccIdentity &)const;{}

RMobilePhone::GetIccMessageWaitingIndicators(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::GetLockInfo(TRequestStatus &,TMobilePhoneLock,TDes8 &)const;{}

RMobilePhone::GetMmsConfig(TRequestStatus &,const TMmsConnParams,TDes8 &)const;{}

RMobilePhone::GetNITZInfo(TMobilePhoneNITZ &)const;{}

RMobilePhone::GetNetworkSelectionSetting(TDes8 &)const;{}

RMobilePhone::GetPersonalisationStatus(TRequestStatus &,TUint32 &)const;{}

RMobilePhone::GetPhoneId(TRequestStatus &,TMobilePhoneIdentityV1 &)const;{}

RMobilePhone::GetPhoneStoreInfo(TRequestStatus &,TDes8 &,const TDesC &)const;{}

RMobilePhone::GetPhoneStoreInfo(TRequestStatus &,TDes8 &,const TDesC &,const TDesC &)const;{}

RMobilePhone::GetScFileInfo(TRequestStatus &,const TScFilePath &,TDes8 &)const;{}

RMobilePhone::GetServiceProviderName(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::GetSubscriberId(TRequestStatus &,TMobilePhoneSubscriberId &)const;{}

RMobilePhone::NotifyCurrentNetworkChange(TRequestStatus &,TDes8 &,TMobilePhoneLocationAreaV1 &)const;{Location}

RMobilePhone::NotifyDefaultPrivacyChange(TRequestStatus &,TMobilePhonePrivacy &)const;{Network-Control}

RMobilePhone::NotifyIccMessageWaitingIndicatorsChange(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::NotifyLockInfoChange(TRequestStatus &,TMobilePhoneLock &,TDes8 &)const;{}

RMobilePhone::NotifyMmsConfig(TRequestStatus &,const TMmsConnParams,TDes8 &)const;{}

RMobilePhone::NotifyMmsUpdate(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::NotifyNITZInfoChange(TRequestStatus &,TMobilePhoneNITZ &)const;{}

RMobilePhone::NotifyNetworkSelectionSettingChange(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::ReadScFile(TRequestStatus &,const TScFilePathWithAccessOffsets &,TDes8 &)const;{}

RMobileUssdMessaging::ReceiveMessage(TRequestStatus &,TDes8 &,TDes8 &)const;{NetworkControl}

RPacketContext::EnumeratePacketFilters(TRequestStatus &,TInt &)const;{}

RPacketContext::GetConfig(TRequestStatus &,TDes8 &)const;{}

RPacketContext::GetDnsInfo(TRequestStatus &,TDes8 &)const;{}

RPacketContext::GetPacketFilterInfo(TRequestStatus &,TInt,TDes8 &)const;{NetworkServices}

RPacketContext::NotifyConfigChanged(TRequestStatus &,TDes8 &)const;{}

RPacketQoS::GetProfileParameters(TRequestStatus &,TDes8 &)const;{}

RPacketQoS::NotifyProfileChanged(TRequestStatus &,TDes8 &)const;{}

RPacketService::EnumerateContextsInNif(TRequestStatus &,const TDesC &,TInt &)const;{}

RPacketService::GetContextNameInNif(TRequestStatus &,const TDesC &,TInt,TDes &)const;{}

RPacketService::GetDefaultContextParams(TDes8 &)const;{}

RPacketService::GetDefaultContextParams(TRequestStatus &,TDes8 &)const;{}

RPacketService::GetNifInfo(TRequestStatus &,TInt,TDes8 &)const;{}

RSat::GetProvisioningRefFile(TRequestStatus &,const TProvisioningFileRef &,TDes8 &)const;{}

RSat::MenuSelection(TRequestStatus &,const TDesC8 &)const;{NetworkControl}

RSat::NotifyCallControlRequest(TRequestStatus &,TDes8 &)const;{NetworkControl}

RSat::NotifyDeclareServicePCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyDisplayTextPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyGetServiceInfoPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyLaunchBrowserPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyLocalInfoPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyMoSmControlRequest(TRequestStatus &,TDes8 &)const;{NetworkControl}

RSat::NotifyOpenChannelPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyPerformCardApduPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyRunAtCommandPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySendDataPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySendDtmfPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySendSmPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySendSsPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySendUssdPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifyServiceSearchPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySetUpIdleModeTextPCmd(TRequestStatus &,TDes8 &)const;{}

RSyncMLDevManProfile::OpenL(RSyncMLSession &,TSmlProfileId);{}

User::MachineConfiguration(TDes8 &,TInt &);{}

Capability: ReadUserData

AgnEntryStorer::StoreEntryL(CStreamStore &,CAgnEntry *);{}

CAgnAnniv::CopyFromL(CAgnEntry *,const MPictureFactory *);{}

CAgnAppt::CopyFromL(CAgnEntry *,const MPictureFactory *);{}

CAgnAppt::CopyFromL(CAgnEntry *,const MPictureFactory *,TCopyHow);{}

CAgnEntry::CopyBasicDetailsFromL(CAgnEntry *,const MPictureFactory *,TCopyHow);{}

CAgnEntry::GetGsDataL();{}

CAgnEntry::LoadAllComponentsL(const MPictureFactory *);{}

CAgnEntry::NotesTextL()const;{}

CAgnEntry::OpenEmbeddedStoreL();{}

CAgnEntry::UpdateNotesTextL();{WriteUserData}

CAgnEntryManager::UpdateEntryL(CAgnEntry *,TStreamId &);{}

CAgnEntryModel::AddEntryL(CAgnEntry *,TAgnEntryId);{WriteUserData}

CAgnEntryModel::AddTodoListL(CAgnTodoList *,TInt);{WriteUserData}

CAgnEntryModel::BuildTodoListsL();{}

CAgnEntryModel::CategoryL(TInt)const;{}

CAgnEntryModel::ChangeTodoListOrderL(TInt,TInt);{WriteUserData}

CAgnEntryModel::ChangeTodoOrderL(CAgnTodoList *,TAgnEntryId,TAgnEntryId);{WriteUserData}

CAgnEntryModel::ChangeTodoOrderL(TAgnTodoListId,TAgnEntryId,TAgnEntryId);{WriteUserData}

CAgnEntryModel::CheckNotifier();{WriteUserData}

CAgnEntryModel::CutEntryL(CAgnEntry *);{WriteUserData}

CAgnEntryModel::CutEntryL(TAgnEntryId);{WriteUserData}

CAgnEntryModel::DeleteEntryL(CAgnEntry *);{WriteUserData}

CAgnEntryModel::DeleteEntryL(TAgnEntryId);{WriteUserData}

CAgnEntryModel::DeleteTodoListL(CAgnTodoList *);{WriteUserData}

CAgnEntryModel::DeleteTodoListL(TAgnTodoListId);{WriteUserData}

CAgnEntryModel::DoChangeTodoOrderL(CAgnTodoList *,TAgnEntryId,TAgnEntryId);{WriteUserData}

CAgnEntryModel::DoDeleteTodoListL(CAgnTodoList *);{}

CAgnEntryModel::DoSaveTodoListsL(CStreamStore &);{WriteUserData}

CAgnEntryModel::DoUpdateEntryL(CAgnEntry *,TAgnEntryId,TBool &,TCommit);{}

CAgnEntryModel::ExportVCalL(TInt,RWriteStream &,CArrayFixFlat< TAgnEntryId > *,const Versit::TVersitCharSet);{}

CAgnEntryModel::FetchAllRelatedGsEntrysL(const HBufC8 &)const;{}

CAgnEntryModel::FetchAllRelatedGsEntrysL(const HBufC8 &,TTime)const;{}

CAgnEntryModel::FetchEntryL(TAgnEntryId)const;{}

CAgnEntryModel::FetchEntryL(TAgnGlobalId)const;{}

CAgnEntryModel::FetchEntryL(TAgnUniqueId)const;{}

CAgnEntryModel::FetchGsEntryL(const HBufC8 &)const;{}

CAgnEntryModel::FetchGsEntryL(const HBufC8 &,TTime)const;{}

CAgnEntryModel::GetGuidL(const CAgnEntry &)const;{}

CAgnEntryModel::GetLiteEntryFromServerL(TAgnEntryId,CAgnSortEntryAllocator *)const;{}

CAgnEntryModel::GetRecurrenceIdL(const CAgnEntry &)const;{}

CAgnEntryModel::OpenL(const TDesC &,TTimeIntervalMinutes,TTimeIntervalMinutes,TTimeIntervalMinutes);{}

CAgnEntryModel::PasteEntryL(CAgnEntry *,TAgnEntryId);{WriteUserData}

CAgnEntryModel::PopulateTodoListNamesL(CAgnTodoListNames *)const;{}

CAgnEntryModel::SetReplicatedEntryAsDeleted(CAgnEntry *);{}

CAgnEntryModel::SetServer(RAgendaServ *);{}

CAgnEntryModel::UpdateEntryL(CAgnEntry *,TAgnEntryId);{WriteUserData}

CAgnEntryModel::UpdateTodoListL(CAgnTodoList *);{WriteUserData}

CAgnEntryStore::UpdateEntryL(const CAgnEntry *,TBool &);{}

CAgnEvent::CopyFromL(CAgnEntry *,const MPictureFactory *);{}

CAgnIndexedModel::BuildIndexL(MAgnProgressCallBack *,TBool,TOpenCallBackFrequency);{}

CAgnIndexedModel::DeleteTidiedEntriesL();{WriteUserData}

CAgnIndexedModel::DoTidyByDateStepL();{WriteUserData}

CAgnIndexedModel::DoTidyByTodoListStepL();{WriteUserData}

CAgnIndexedModel::OpenL(const TDesC &,TTimeIntervalMinutes,TTimeIntervalMinutes,TTimeIntervalMinutes);{}

CAgnIndexedModel::OpenL(const TDesC &,TTimeIntervalMinutes,TTimeIntervalMinutes,TTimeIntervalMinutes,MAgnProgressCallBack *,TBool,TOpenCallBackFrequency);{}

CAgnIndexedModel::SetUpTidyByDateL(const TAgnFilter &,const TTime &,CStreamStore *,TStreamId &,TTidyDirective);{WriteUserData}

CAgnIndexedModel::SetUpTidyByTodoListL(CStreamStore *,TStreamId &,TTidyDirective);{WriteUserData}

CAgnIndexedModel::TidyByDateL(const TAgnFilter &,const TTime &,const TTime &,const TTime &,MAgnProgressCallBack *,TTidyDirective);{WriteUserData}

CAgnIndexedModel::TidyByTodoListL(const CArrayFixFlat< TAgnTodoListId > *,MAgnProgressCallBack *,TTidyDirective,TTidyTodoListHow);{WriteUserData}

CAgnModel::AddTodoListL(CAgnTodoList *,TInt);{WriteUserData}

CAgnModel::CutInstanceL(CAgnEntry *,TAgnWhichInstances);{WriteUserData}

CAgnModel::CutInstanceL(const TAgnInstanceId &,TAgnWhichInstances);{WriteUserData}

CAgnModel::DeleteInstanceL(CAgnEntry *,TAgnWhichInstances);{WriteUserData}

CAgnModel::DeleteTodoListL(CAgnTodoList *);{WriteUserData}

CAgnModel::DeleteTodoListL(TAgnTodoListId);{WriteUserData}

CAgnModel::PopulateTodoInstanceListL(CAgnTodoInstanceList *,const TTime &)const;{}

CAgnModel::UpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{WriteUserData}

CAgnModel::UpdateTodoListL(CAgnTodoList *);{WriteUserData}

CAgnTodo::CopyFromL(CAgnEntry *,const MPictureFactory *);{}

CAgnTodo::SetDuration(TTimeIntervalDays);{}

CCalCategoryManager::FilterCategoryL(const CCalCategory &,RPointerArray< CCalEntry > &,MCalProgressCallBack &);{}

CCalDataExchange::ExportL(TUid,RWriteStream &,RPointerArray< CCalEntry > &);{}

CCalEntryView::FetchL(const TDesC8 &,RPointerArray< CCalEntry > &)const;{}

CCalInstanceView::FindInstanceL(RPointerArray< CCalInstance > &,CalCommon::TCalViewFilter,const CalCommon::TCalTimeRange &)const;{}

CCalInstanceView::FindInstanceL(RPointerArray< CCalInstance > &,CalCommon::TCalViewFilter,const CalCommon::TCalTimeRange &,const TCalSearchParams &)const;{}

CCalSession::OpenL(const TDesC &)const;{}

CContactConcatenatedView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactDatabase::CommitContactL(const CContactItem &);{WriteUserData}

CContactDatabase::ContactDatabaseExistsL(const TDesC &);{}

CContactDatabase::DatabaseDrive(TDriveUnit &);{}

CContactDatabase::DefaultContactDatabaseExistsL();{}

CContactDatabase::DeleteContactL(TContactItemId);{WriteUserData}

CContactDatabase::DeleteContactsL(const CContactIdArray &);{WriteUserData}

CContactDatabase::GetCurrentDatabase(TDes &)const;{}

CContactDatabase::GetCurrentItem()const;{}

CContactDatabase::GetDefaultNameL(TDes &);{}

CContactDatabase::GetSpeedDialFieldL(TInt,TDes &);{}

CContactDatabase::ListDatabasesL();{}

CContactDatabase::ListDatabasesL(TDriveUnit);{}

CContactDatabase::MatchPhoneNumberL(const TDesC &,TInt);{}

CContactDatabase::Open(TRequestStatus &,TThreadAccess);{}

CContactDatabase::Open(const TDesC &,TRequestStatus &,TThreadAccess);{}

CContactDatabase::OpenL(TThreadAccess);{}

CContactDatabase::OpenL(const TDesC &,TThreadAccess);{}

CContactDatabase::OpenTablesL();{}

CContactDatabase::RemoveSpeedDialAttribsFromContactL(TContactItemId,TInt);{WriteUserData}

CContactDatabase::RemoveSpeedDialFieldL(TContactItemId,TInt);{WriteUserData}

CContactDatabase::SetFieldAsSpeedDialL(CContactItem &,TInt,TInt);{WriteUserData}

CContactDatabase::UnfiledContactsL();{}

CContactDatabase::UpdateContactLC(TContactItemId,CContactItem *);{WriteUserData}

CContactDatabase::doCommitContactL(const CContactItem &,TBool,TBool);{WriteUserData}

CContactDatabase::doDeleteContactL(TContactItemId,TBool,TBool,TBool);{WriteUserData}

CContactDatabase::doDeleteContactsL(const CContactIdArray &,TBool &);{WriteUserData}

CContactFilteredView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactFindView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactGroupView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactLocalView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactNamedRemoteView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactNamedRemoteView::ChangeSortOrderL(const RContactViewSortOrder &);{}

CContactNamedRemoteView::ConstructL(MContactViewObserver &,const TDesC &,const RContactViewSortOrder &,TContactViewPreferences);{}

CContactNamedRemoteView::ConstructL(MContactViewObserver &,const TDesC &,const RContactViewSortOrder &,TContactViewPreferences,const TDesC8 &);{}

CContactNamedRemoteView::NewL(MContactViewObserver &,const TDesC &,const CContactDatabase &,const RContactViewSortOrder &,TContactViewPreferences);{}

CContactNamedRemoteView::NewL(MContactViewObserver &,const TDesC &,const CContactDatabase &,const RContactViewSortOrder &,TContactViewPreferences,const TDesC8 &);{}

CContactRemoteView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactRemoteView::ConstructL(MContactViewObserver &,const RContactViewSortOrder &,TContactViewPreferences);{}

CContactRemoteView::ConstructL(MContactViewObserver &,const RContactViewSortOrder &,TContactViewPreferences,const TDesC8 &);{}

CContactRemoteView::GetSortOrderL(RContactViewSortOrder &);{}

CContactRemoteView::NewL(MContactViewObserver &,const CContactDatabase &,const RContactViewSortOrder &,TContactViewPreferences);{}

CContactRemoteView::NewL(MContactViewObserver &,const CContactDatabase &,const RContactViewSortOrder &,TContactViewPreferences,const TDesC8 &);{}

CContactRemoteViewBase::AllFieldsLC(TInt,const TDesC &)const;{}

CContactRemoteViewBase::AtL(TInt)const;{}

CContactRemoteViewBase::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactRemoteViewBase::ConstructL(MContactViewObserver &);{}

CContactRemoteViewBase::ContactAtL(TInt)const;{}

CContactRemoteViewBase::ContactViewPreferences();{}

CContactRemoteViewBase::ContactsMatchingCriteriaL(const MDesC16Array &,RPointerArray< CViewContact > &);{}

CContactRemoteViewBase::ContactsMatchingPrefixL(const MDesC16Array &,RPointerArray< CViewContact > &);{}

CContactRemoteViewBase::CountL()const;{}

CContactRemoteViewBase::FindL(TContactItemId)const;{}

CContactRemoteViewBase::GetContactIdsL(const CArrayFix< TInt > &,CContactIdArray &);{}

CContactRemoteViewBase::GetContactsMatchingFilterL(TInt,RArray< TContactIdWithMapping > &);{}

CContactRemoteViewBase::SortOrderL()const;{}

CContactSubView::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactViewBase::AllFieldsLC(TInt,const TDesC &)const=0;{}

CContactViewBase::AtL(TInt)const=0;{}

CContactViewBase::CContactViewBase_Reserved_1(TFunction,TAny *);{}

CContactViewBase::ContactAtL(TInt)const=0;{}

CContactViewBase::ContactViewPreferences()=0;{}

CContactViewBase::ContactsMatchingCriteriaL(const MDesC16Array &,RPointerArray< CViewContact > &);{}

CContactViewBase::ContactsMatchingPrefixL(const MDesC16Array &,RPointerArray< CViewContact > &);{}

CContactViewBase::CountL()const=0;{}

CContactViewBase::FindL(TContactItemId)const=0;{}

CContactViewBase::GetContactIdsL(const CArrayFix< TInt > &,CContactIdArray &);{}

CContactViewBase::SortOrderL()const=0;{}

CFaxTransfer::Start(TRequestStatus &);{NetworkServices, WriteUserData}

CFaxTransfer::Stop();{NetworkServices, WriteUserData}

CRetrieveMobilePhoneBroadcastIdList::RetrieveListL();{}

CRetrieveMobilePhoneBroadcastIdList::Start(TRequestStatus &,RMobileBroadcastMessaging::TMobile-BroadcastIdType);{}

CRetrieveMobilePhoneENList::RetrieveListL();{}

CRetrieveMobilePhoneENList::Start(TRequestStatus &);{}

CRetrieveMobilePhoneONList::RetrieveListL();{}

CRetrieveMobilePhoneONList::Start(TRequestStatus &);{}

CRetrieveMobilePhoneSmsList::RetrieveCdmaListL();{}

CRetrieveMobilePhoneSmsList::RetrieveGsmListL();{}

CRetrieveMobilePhoneSmsList::Start(TRequestStatus &);{}

CRetrieveMobilePhoneSmsList::StartBatch(TRequestStatus &,TInt,TInt);{}

CSWICertStore::Retrieve(const CCTCertInfo &,TDes8 &,TRequestStatus &);{}

CSmsEventLogger::GetEvent(TRequestStatus &,TLogId);{}

CSmsHeader::InternalizeL(RMsvReadStream &);{}

CSmsHeader::RestoreL(CMsvStore &);{}

CSmsHeader::RestoreL(CMsvStore &,CEditableText &);{}

CTelephony::GetCallInfo(TDes8 &,TDes8 &,TDes8 &)const;{}

CTelephony::GetCurrentNetworkInfo(TRequestStatus &,TDes8 &)const;{Location, ReadDeviceData}

CTelephony::GetCurrentNetworkName(TRequestStatus &,TDes8 &)const;{}

CTelephony::GetOperatorName(TRequestStatus &,TDes8 &)const;{}

CUnifiedCertStore::Retrieve(const CCTCertInfo &,CCertificate *&,TRequestStatus &);{}

CUnifiedCertStore::Retrieve(const CCTCertInfo &,TDes8 &,TRequestStatus &);{}

CUnifiedKeyStore::ExportEncryptedKey(TCTTokenObjectHandle,const CPBEncryptParms &,HBufC8 *&,TRequestStatus &);{}

CUnifiedKeyStore::ExportKey(TCTTokenObjectHandle,HBufC8 *&,TRequestStatus &);{}

CUnifiedKeyStore::GetKeyInfo(TCTTokenObjectHandle,CCTKeyInfo *&,TRequestStatus &);{}

CUnifiedKeyStore::List(RMPointerArray< CCTKeyInfo > &,const TCTKeyAttributeFilter &,TRequestStatus &);{}

CUnifiedKeyStore::Open(const TCTTokenObjectHandle &,MCTDH *&,TRequestStatus &);{}

CUnifiedKeyStore::Open(const TCTTokenObjectHandle &,MCTDecryptor *&,TRequestStatus &);{}

CUnifiedKeyStore::Open(const TCTTokenObjectHandle &,MDSASigner *&,TRequestStatus &);{}

CUnifiedKeyStore::Open(const TCTTokenObjectHandle &,MRSASigner *&,TRequestStatus &);{}

MCertStore::Retrieve(const CCTCertInfo &,TDes8 &,TRequestStatus &)=0;{}

MKeyStore::List(RMPointerArray< CCTKeyInfo > &,const TCTKeyAttributeFilter &,TRequestStatus &)=0;{}

RAgendaServ::AddEntryL(CAgnEntry *,TAgnEntryId,TInt,TBool);{WriteUserData}

RAgendaServ::CategoryL(TInt);{}

RAgendaServ::CreateTransmitBufferL(TInt)const;{}

RAgendaServ::DateIteratorCurrentElement(CAgnSortEntryAllocator *);{}

RAgendaServ::DeleteEntryL(CAgnEntry *);{WriteUserData}

RAgendaServ::FetchEntryL(TAgnEntryId,CParaFormatLayer *,CCharFormatLayer *)const;{}

RAgendaServ::FetchEntryL(TAgnGlobalId,CParaFormatLayer *,CCharFormatLayer *)const;{}

RAgendaServ::FetchEntryL(TAgnUniqueId,CParaFormatLayer *,CCharFormatLayer *)const;{}

RAgendaServ::GetEmbeddedReadStreamL(TStreamId);{}

RAgendaServ::GetLiteEntryL(TAgnEntryId,CAgnSortEntryAllocator *)const;{}

RAgendaServ::GetReadStreamL(TStreamId);{}

RAgendaServ::RefreshTodoListListL(CAgnTodoListList *,CAgnDeletedTodoListList *);{}

RAgendaServ::RestoreNotesTextL(TStreamId);{}

RAgendaServ::UpdateEntryL(CAgnEntry *,TAgnEntryId);{WriteUserData}

RCdmaMobileCall::NotifyIncomingNetworkFlashWithInfo(TRequestStatus &,TDes8 &)const;{Network-Services}

RCdmaMobilePhone::NotifyIncomingInfoRecord(TRequestStatus &,TDes8 &)const;{}

RContactRemoteView::AllFieldsLC(TInt,const TDesC &)const;{}

RContactRemoteView::AtL(TInt)const;{}

RContactRemoteView::ChangeSortOrderL(const RContactViewSortOrder &);{}

RContactRemoteView::ContactAtL(TInt);{}

RContactRemoteView::ContactViewPreferencesL();{}

RContactRemoteView::ContactsMatchingCriteriaL(const MDesC16Array &,RPointerArray< CViewContact > &,TBool,TUid);{}

RContactRemoteView::CountL()const;{}

RContactRemoteView::FindL(TContactItemId)const;{}

RContactRemoteView::GetContactIdsL(const CArrayFix< TInt > &,CContactIdArray &);{}

RContactRemoteView::GetContactsMatchingFilterL(TInt,RArray< TContactIdWithMapping > &);{}

RContactRemoteView::GetSortOrderL(RContactViewSortOrder &);{}

RContactRemoteView::OpenL(const CContactDatabase &,const RContactViewSortOrder &,TContactViewPreferences,const TUid &,const TDesC8 &);{}

RContactRemoteView::OpenL(const CContactDatabase &,const TDesC &,const RContactViewSortOrder &,TContactViewPreferences,const TUid &,const TDesC8 &);{}

RContactRemoteView::RequestViewEvent(TPckgBuf< TContactViewEvent > &,TRequestStatus &);{}

RContactRemoteView::SortOrderL();{}

RFax::GetProgress(TProgress &);{}

RFax::Read(TRequestStatus &,TDes8 &);{NetworkServices}

RMobileBroadcastMessaging::GetFilterSetting(TMobilePhoneBroadcastFilter &)const;{}

RMobileBroadcastMessaging::GetLanguageFilter(TRequestStatus &,TDes16 &)const;{}

RMobileBroadcastMessaging::NotifyFilterSettingChange(TRequestStatus &,TMobilePhoneBroadcastFilter &)const;{}

RMobileBroadcastMessaging::NotifyLanguageFilterChange(TRequestStatus &,TDes16 &)const;{}

RMobileCall::GetMobileCallInfo(TDes8 &)const;{}

RMobileCall::NotifyRemotePartyInfoChange(TRequestStatus &,TDes8 &)const;{}

RMobileCall::ReceiveUUI(TRequestStatus &,TMobileCallUUI &)const;{NetworkServices}

RMobilePhone::GetAirTimeDuration(TTimeIntervalSeconds &)const;{}

RMobilePhone::GetCostInfo(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::GetCurrentNetwork(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::GetMailboxNumbers(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::NotifyAirTimeDurationChange(TRequestStatus &,TTimeIntervalSeconds &)const;{}

RMobilePhone::NotifyCostInfoChange(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::NotifyCurrentNetworkChange(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::NotifyMailboxNumbersChange(TRequestStatus &,TDes8 &)const;{}

RMobilePhoneBookStore::Read(TRequestStatus &,TInt,TInt,TDes8 &)const;{}

RMobilePhoneStore::GetInfo(TRequestStatus &,TDes8 &)const;{}

RMobilePhoneStore::NotifyStoreEvent(TRequestStatus &,TUint32 &,TInt &)const;{}

RMobilePhoneStore::Read(TRequestStatus &,TDes8 &)const;{}

RMobileSmsMessaging::ReceiveMessage(TRequestStatus &,TDes8 &,TDes8 &)const;{NetworkServices}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::CopyEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp);{None, WriteUserData}

RMsvServerSession::CopyEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp,TRequestStatus &);{None, WriteUserData}

RMsvServerSession::CreateAttachmentForWriteL(TMsvId,TDes &,RFile &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::DecStoreReaderCount(TMsvId);{None}

RMsvServerSession::DeleteAttachment(TMsvId,const TDesC &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp,TRequestStatus &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::DeleteFileStoreL(TMsvId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::GetChildIdsL(TMsvId,const CMsvEntryFilter &,CMsvEntrySelection &);{None}

RMsvServerSession::GetChildren(TMsvId,CArrayPtrFlat< CMsvClientEntry > &,const TMsvSelectionOrdering &);{None}

RMsvServerSession::GetEntry(TMsvId,TMsvId &,TMsvEntry &);{None}

RMsvServerSession::LockStore(TMsvId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::MoveEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp);{None, WriteUserData}

RMsvServerSession::MoveEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp,TRequestStatus &);{None, WriteUserData}

RMsvServerSession::OpenAttachmentForWriteL(TMsvId,const TDesC &,RFile &);{None, WriteUserData}

RMsvServerSession::OpenAttachmentL(TMsvId,const TDesC &,RFile &);{None}

RMsvServerSession::OpenFileStoreForRead(TMsvId,RFile &);{None}

RMsvServerSession::OpenTempStoreFileL(TMsvId,RFile &);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::ReadStore(TMsvId);{None}

RMsvServerSession::ReleaseStore(TMsvId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::RemoveEntry(TMsvId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::ReplaceFileStoreL(TMsvId);{None, WriteDeviceData, WriteUserData}

RMsvServerSession::TransferCommandL(const CMsvEntrySelection &,TInt,const TDesC8 &,TMsvOp);{WriteUserData}

RMsvServerSession::TransferCommandL(const CMsvEntrySelection &,TInt,const TDesC8 &,TMsvOp,TRequestStatus &);{WriteUserData}

RPacketContext::GetDataVolumeTransferred(TDataVolume &)const;{}

RPacketContext::GetDataVolumeTransferred(TRequestStatus &,TDataVolume &)const;{}

RPacketContext::NotifyDataTransferred(TRequestStatus &,TDataVolume &,TUint,TUint)const;{}

RPhoneBookSession::CancelRequest(TPhonebookSyncRequestCancel,TUid);{WriteUserData}

RPhoneBookSession::DoSynchronisation(TRequestStatus &);{WriteUserData}

RPhoneBookSession::DoSynchronisation(TRequestStatus &,TUid);{WriteUserData}

RPhoneBookSession::ShutdownServer(TBool);{WriteUserData}

RPhoneBookSession::ValidateContact(MContactSynchroniser::TValidateOperation,TContactItemId);{}

RSat::NotifyCbDownload(TRequestStatus &,TDes8 &)const;{NetworkControl}

RSat::NotifySetUpCallPCmd(TRequestStatus &,TDes8 &)const;{}

RSat::NotifySmsPpDownload(TRequestStatus &,TDes8 &)const;{NetworkControl}

RWorldServer::Home(TWorldId &)const;{}

TAgnInstanceEditor::CreateAndStoreExceptionL(CAgnEntry *&,CAgnEntry *,TAgnEntryId);{WriteUserData}

TAgnInstanceEditor::DeleteInstanceL(CAgnEntry *,TAgnWhichInstances);{WriteUserData}

TAgnInstanceEditor::DoUpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{WriteUser-Data}

TAgnInstanceEditor::SplitRepeatL(CAgnEntry *&,TAgnWhichInstances,CAgnEntry *,TAgnEntryId);{Wri-teUserData}

TAgnInstanceEditor::UpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{WriteUserData}

Capability: SurroundingsDD

Capability: SwEvent

RWindowGroup::CaptureKey(TUint,TUint,TUint);{}

RWindowGroup::CaptureKey(TUint,TUint,TUint,TInt);{}

RWindowGroup::CaptureKeyUpAndDowns(TUint,TUint,TUint);{}

RWindowGroup::CaptureKeyUpAndDowns(TUint,TUint,TUint,TInt);{}

RWindowGroup::CaptureLongKey(TTimeIntervalMicroSeconds32,TUint,TUint,TUint,TUint,TInt,TUint);{}

RWindowGroup::CaptureLongKey(TUint,TUint,TUint,TUint,TInt,TUint);{}

RWsSession::SendEventToAllWindowGroups(TInt,const TWsEvent &);{}

RWsSession::SendEventToAllWindowGroups(const TWsEvent &);{}

RWsSession::SendEventToOneWindowGroupsPerClient(const TWsEvent &);{}

RWsSession::SendEventToWindowGroup(TInt,const TWsEvent &);{}

RWsSession::SimulateKeyEvent(TKeyEvent);{}

RWsSession::SimulateRawEvent(TRawEvent);{}

UserSvr::AddEvent(const TRawEvent &);{}

Capability: Tcb

RLocalDrive::Connect(TInt,TBool &);{}

TBusLocalDrive::Connect(TInt,TBool &);{}

Capability: TrustedUI

Capability: UserEnvironment

CCamera::NewDuplicateL(MCameraObserver &,TInt);{}

CCamera::NewDuplicateL(MCameraObserver2 &,TInt);{}

CCamera::NewL(MCameraObserver &,TInt);{}

CCamera::NewL(MCameraObserver2 &,TInt,TInt);{MultimediaDD}

CMMFDevSound::RecordData();{}

CMMFDevSound::RecordInitL();{}

CMdaAudioInputStream::NewL(MMdaAudioInputStreamCallback &);{}

CMdaAudioInputStream::NewL(MMdaAudioInputStreamCallback &,TInt,TMdaPriorityPreference);{Multi-mediaDD}

CMdaAudioRecorderUtility::RecordL();{}

Capability: WriteDeviceData

CCertificateAppInfoManager::AddL(const TCertificateAppInfo &);{}

CCertificateAppInfoManager::RemoveL(const TUid &);{}

CCoeEnv::InstallFepL(TUid);{}

CCoeEnv::InstallFepL(TUid,const TBool);{}

CCoeFep::IsTurnedOffByL(const TKeyEvent &)const;{}

CCoeFep::IsTurnedOnByL(const TKeyEvent &)const;{}

CCoeFep::WriteAttributeDataAndBroadcastL(TUid);{}

CCoeFep::WriteAttributeDataAndBroadcastL(const TArray< TUid > &);{}

CEikonEnv::UpdateSystemColorListL(const CColorList &);{}

CFepGenericGlobalSettings::StoreChangesAndBroadcastL();{}

CLogClient::AddEventType(const CLogEventType &,TRequestStatus &);{}

CLogClient::ChangeConfig(const TLogConfig &,TRequestStatus &);{}

CLogClient::ChangeEventType(const CLogEventType &,TRequestStatus &);{}

CLogClient::ClearLog(TInt,TRequestStatus &);{}

CLogClient::ClearLog(const TTime &,TRequestStatus &);{}

CLogClient::DeleteEventType(TUid,TRequestStatus &);{}

CLogViewDuplicate::RemoveL(TLogId);{}

CLogViewDuplicate::RemoveL(TRequestStatus &);{}

CLogViewRecent::ClearDuplicatesL();{}

CLogViewRecent::RemoveL(TLogId);{}

CLogViewRecent::RemoveL(TRequestStatus &);{}

CMobilePhoneStoredNetworkList::ChangeEntryL(TInt,const RMobile-
Phone::TMobilePreferredNetworkEntryV3 &);{}

CMobilePhoneStoredNetworkList::DeleteEntryL(TInt);{}

CMobilePhoneStoredNetworkList::InsertEntryL(TInt,const RMobile-
Phone::TMobilePreferredNetworkEntryV3 &);{}

CServiceRegistry::RemoveEntry(TUid,const TDataType &);{}

CServiceRegistry::SetDefault(TUid,const TDataType &,TUid);{}

CSmsMessageSettings::ExternalizeL(RWriteStream &)const;{}

CSmsSimParamOperation::DoWriteSimParamsL(const CMobilePhoneSmspList &);{}

CSmsSimParamOperation::StoreSimParamsL(CMsvStore &,const CMobilePhoneSmspList &);{}

CSmsSimParamOperation::TransferCommandL(TInt);{ReadDeviceData}

CSmsSimParamOperation::WriteSimParamsL(const CMobilePhoneSmspList &,TUid,TMsvId,CMsvSession &,TRequestStatus &);{}

CUnifiedCertStore::Remove(const CCTCertInfo &,TRequestStatus &);{WriteUserData}

CUnifiedCertStore::SetApplicability(const CCTCertInfo &,const RArray< TUid > &,TRequestStatus &);{}

CUnifiedCertStore::SetTrust(const CCTCertInfo &,TBool,TRequestStatus &);{}

CWsScreenDevice::SetCurrentRotations(TInt,CFbsBitGc::TGraphicsOrientation)const;{}

CWsScreenDevice::SetCustomPalette(const CPalette *);{}

CWsScreenDevice::SetScreenMode(TInt);{}

CWsScreenDevice::SetScreenModeEnforcement(TScreenModeEnforcement)const;{}

MCTWritableCertStore::Add(const TDesC &,TCertificateFormat,TCertificateOwnerType,const TKeyIdentifier *,const TKeyIdentifier *,const TDesC8 &,TRequestStatus &)=0;{WriteUserData}

MCTWritableCertStore::Add(const TDesC &,TCertificateFormat,TCertificateOwnerType,const TKeyIdentifier *,const TKeyIdentifier *,const TDesC8 &,const TBool,TRequestStatus &);{WriteUserData}

MCTWritableCertStore::Remove(const CCTCertInfo &,TRequestStatus &)=0;{WriteUserData}

MCTWritableCertStore::SetApplicability(const CCTCertInfo &,const RArray< TUid > &,TRequestStatus &)=0;{}

MCTWritableCertStore::SetTrust(const CCTCertInfo &,TBool,TRequestStatus &)=0;{}

MFepAttributeStorer::WriteAttributeDataAndBroadcastL(CCoeEnv &,TUid);{}

MFepAttributeStorer::WriteAttributeDataAndBroadcastL(CCoeEnv &,const TArray< TUid > &);{}

RApaLsSession::DeleteDataMapping(const TDataType &);{}

RApaLsSession::InsertDataMapping(const TDataType &,TDataTypePriority,TUid);{}

RApaLsSession::InsertDataMappingIfHigher(const TDataType &,TDataTypePriority,TUid,TBool &);{}

RApaLsSession::SetAcceptedConfidence(TInt);{}

RBTLocalDevice::Modify(const TBTLocalDevice &);{LocalServices}

RBTLocalDevice::Modify(const TBTLocalDevice &,TRequestStatus &);{LocalServices}

RBTRegistry::DeleteAllInView(TRequestStatus &);{LocalServices}

RBTRegistry::ModifyBluetoothDeviceNameL(const TBTDevAddr &,const TDesC8 &,TRequestStatus &);{LocalServices}

RBTRegistry::ModifyDevice(const TBTNamelessDevice &,TRequestStatus &);{LocalServices}

RBTRegistry::UnpairAllInView(TRequestStatus &);{LocalServices}

RBTRegistry::UnpairDevice(const TBTDevAddr &,TRequestStatus &);{LocalServices}

RCall::SetFaxSettings(const TFaxSessionSettings &)const;{}

RCdmaMobilePhone::LockNam(TRequestStatus &,TMobilePhoneNamCommitStatus)const;{}

RCdmaMobilePhone::ProcessOtaRequest(TRequestStatus &,const TDesC8 &,TDes8 &)const;{Network-Services, ReadDeviceData}

RCdmaMobilePhone::SetDTMFBurstDuration(TRequestStatus &,TMobilePhoneDtmfOnDuration,TMobile-PhoneDtmfOffDuration)const;{}

RCdmaMobilePhone::SetDTMFMode(TRequestStatus &,TMobilePhoneDtmfMode)const;{}

RCdmaMobilePhone::SetLocationPrivacy(TRequestStatus &,TMobilePhoneLocationPrivacy)const;{}

RCdmaMobilePhone::SetOtaSettings(TRequestStatus &,TUint)const;{}

RCdmaMobilePhone::SetTtyMode(TRequestStatus &,TMobilePhoneTtyMode)const;{}

RCdmaMobilePhone::StartOta(TRequestStatus &,TOtaServiceType,const TDes8 &)const;{NetworkServ-ices}

RCdmaMobilePhone::StopOta(TRequestStatus &,TOtaServiceType)const;{NetworkServices}

RCdmaMobilePhone::StorePreferredLanguagesListL(TRequestStatus &,CMobilePhonePreferredLan-guagesList *)const;{}

RCdmaMobilePhone::UnlockNam(TRequestStatus &,const RMobilePhone::TMobilePassword &)const;{}

RCdmaMobilePhone::WriteOtaStoreBlock(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{}

RDmDomain::Connect(TDmDomainId);{}

RMobileCall::ActivateUUS(TRequestStatus &,const TDesC8 &)const;{NetworkServices}

RMobileNamStore::SetActiveNam(TRequestStatus &,TInt)const;{}

RMobileNamStore::StoreAllL(TRequestStatus &,TInt,CMobilePhoneNamList *)const;{}

RMobileNamStore::StoreAllL(TRequestStatus &,TInt,CMobilePhoneNamListV4 *)const;{}

RMobilePhone::AbortSecurityCode(TMobilePhoneSecurityCode)const;{}

RMobilePhone::ChangeSecurityCode(TRequestStatus &,TMobilePhoneSecurityCode,const TMobilePhonePasswordChangeV1 &)const;{}

RMobilePhone::ClearBlacklist(TRequestStatus &)const;{}

RMobilePhone::ClearCostMeter(TRequestStatus &,TMobilePhoneCostMeters)const;{}

RMobilePhone::DeactivateCCBS(TRequestStatus &,TInt)const;{NetworkServices}

RMobilePhone::InitialiseMM(TRequestStatus &,TDes8 &)const;{NetworkControl}

RMobilePhone::ProgramFeatureCode(TRequestStatus &,const TDesC &,TMobilePhoneNetworkService,TMobilePhoneServiceAction)const;{}

RMobilePhone::SelectNetwork(TRequestStatus &,TBool,const TMobilePhoneNetworkManualSelection &)const;{NetworkServices}

RMobilePhone::SendNetworkServiceRequest(TRequestStatus &,const TDesC &)const;{NetworkServices}

RMobilePhone::SendNetworkServiceRequestNoFdnCheck(TRequestStatus &,const TDesC &)const;{NetworkControl, NetworkServices}

RMobilePhone::SetALSLine(TRequestStatus &,TMobilePhoneALSLine)const;{}

RMobilePhone::SetAlternatingCallMode(TRequestStatus &,TMobilePhoneAlternatingCallMode,TMobileService)const;{}

RMobilePhone::SetCallBarringPassword(TRequestStatus &,const TMobilePhonePasswordChangeV1 &)const;{NetworkServices}

RMobilePhone::SetCallBarringStatus(TRequestStatus &,TMobilePhoneCBCondition,const TMobilePhoneCBChangeV1 &)const;{NetworkServices}

RMobilePhone::SetCallForwardingStatus(TRequestStatus &,TMobilePhoneCFCondition,const TMobilePhoneCFChangeV1 &)const;{NetworkServices}

RMobilePhone::SetCallWaitingStatus(TRequestStatus &,TMobileService,TMobilePhoneServiceAction)const;{NetworkServices}

RMobilePhone::SetDefaultPrivacy(TRequestStatus &,TMobilePhonePrivacy)const;{NetworkControl}

RMobilePhone::SetFdnSetting(TRequestStatus &,TMobilePhoneFdnSetting)const;{}

RMobilePhone::SetIccMessageWaitingIndicators(TRequestStatus &,const TDesC8 &)const;{}

RMobilePhone::SetIdentityServiceStatus(TRequestStatus &,const TMobilePhoneIdService,const TMobilePhoneIdServiceSetting)const;{}

RMobilePhone::SetIncomingCallType(TRequestStatus &,TMobilePhoneIncomingCallType,TDes8 &)const;{}

RMobilePhone::SetLockSetting(TRequestStatus &,TMobilePhoneLock,TMobilePhoneLockSetting)const;{}

RMobilePhone::SetMaxCostMeter(TRequestStatus &,TUint)const;{}

RMobilePhone::SetMmsUserConnParams(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::SetMmsUserPreferences(TRequestStatus &,TDes8 &)const;{}

RMobilePhone::SetMulticallParams(TRequestStatus &,TInt)const;{}

RMobilePhone::SetMultimediaCallPreference(TRequestStatus &,TMobilePhoneMultimediaSettings)const;{}

RMobilePhone::SetNetworkSelectionSetting(TRequestStatus &,const TDes8 &)const;{}

RMobilePhone::SetPersonalisationStatus(TRequestStatus &,const TMobilePhonePersonalisation,const TDes &)const;{NetworkControl}

RMobilePhone::SetPuct(TRequestStatus &,const TDesC8 &)const;{}

RMobilePhone::SetSSPassword(TRequestStatus &,const TDesC8 &,const TInt)const;{NetworkServices}

RMobilePhone::SetSmartCardApplicationStatus(TRequestStatus &,const TAID &,TSmartCardApplicationAction)const;{NetworkControl}

RMobilePhone::SetUSimApplicationStatus(TRequestStatus &,const TAID,TUSimAppAction)const;{}

RMobilePhone::SetUSimAppsSelectionMode(TUSimSelectionMode)const;{}

RMobilePhone::SetUUSSetting(TRequestStatus &,TMobilePhoneUUSSetting)const;{NetworkServices}

RMobilePhone::StorePreferredNetworksListL(TRequestStatus &,CMobilePhoneStoredNetworkList *)const;{}

RMobilePhone::UpdateScFile(TRequestStatus &,const TScFilePathWithAccessOffsets &,TDes8 &)const;{NetworkControl}

RMobilePhone::VerifySecurityCode(TRequestStatus &,TMobilePhoneSecurityCode,const TMobilePassword &,const TMobilePassword &)const;{}

RMobileSmsMessaging::SetMoSmsBearer(TRequestStatus &,TMobileSmsBearer)const;{}

RMobileSmsMessaging::SetReceiveMode(TRequestStatus &,TMobileSmsReceiveMode)const;{}

RMobileSmsMessaging::StoreSmspListL(TRequestStatus &,CMobilePhoneSmspList *)const;{}

RMobileUssdMessaging::SendMessage(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkControl, NetworkServices}

RMobileUssdMessaging::SendMessageNoFdnCheck(TRequestStatus &,const TDesC8 &,const TDesC8 &)const;{NetworkControl, NetworkServices}

RMsvServerSession::ChangeDriveL(TInt,TMsvOp,TRequestStatus &);{}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, ReadUserData, WriteUserData}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, ReadUserData, WriteUserData}

RMsvServerSession::CloseMessageServer();{}

RMsvServerSession::CopyStoreL(const TDriveUnit &,TMsvOp,TRequestStatus &);{}

RMsvServerSession::CreateAttachmentForWriteL(TMsvId,TDes &,RFile &);{None, ReadUserData, WriteUserData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, ReadUserData, WriteUserData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, ReadUserData, WriteUserData}

RMsvServerSession::DeInstallMtmGroup(const TDesC &);{}

RMsvServerSession::DeleteAttachment(TMsvId,const TDesC &);{None, ReadUserData, WriteUserData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp);{None, ReadUserData, WriteUserData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp,TRequestStatus &);{None, ReadUserData, WriteUserData}

RMsvServerSession::DeleteFileStoreL(TMsvId);{None, ReadUserData, WriteUserData}

RMsvServerSession::DeleteStoreL(const TDriveUnit &,TMsvOp,TRequestStatus &);{}

RMsvServerSession::InstallMtmGroup(const TDesC &);{}

RMsvServerSession::LockStore(TMsvId);{None, ReadUserData, WriteUserData}

RMsvServerSession::OpenTempStoreFileL(TMsvId,RFile &);{None, ReadUserData, WriteUserData}

RMsvServerSession::ReleaseStore(TMsvId);{None, ReadUserData, WriteUserData}

RMsvServerSession::RemoveEntry(TMsvId);{None, ReadUserData, WriteUserData}

RMsvServerSession::ReplaceFileStoreL(TMsvId);{None, ReadUserData, WriteUserData}

RPacketContext::AddMediaAuthorizationL(TRequestStatus &,CTFTMediaAuthorizationV3 &)const;{NetworkServices}

RPacketContext::AddPacketFilter(TRequestStatus &,const TDesC8 &)const;{NetworkServices}

RPacketContext::Delete(TRequestStatus &)const;{}

RPacketContext::RemoveMediaAuthorization(TRequestStatus &,TAuthorizationToken &)const;{NetworkServices}

RPacketContext::RemovePacketFilter(TRequestStatus &,TInt)const;{NetworkServices}

RPacketContext::SetConfig(TRequestStatus &,const TDesC8 &)const;{}

RPacketQoS::SetProfileParameters(TRequestStatus &,TDes8 &)const;{NetworkServices}

RPacketService::SetAttachMode(TAttachMode)const;{}

RPacketService::SetAttachMode(TRequestStatus &,TAttachMode)const;{}

RPacketService::SetDefaultContextParams(TRequestStatus &,const TDesC8 &)const;{}

RPacketService::SetDefaultContextParams(const TDesC8 &)const;{}

RPacketService::SetMSClass(TRequestStatus &,TMSClass)const;{NetworkServices}

RPacketService::SetPreferredBearer(TRequestStatus &,TPreferredBearer)const;{NetworkServices}

RProperty::Define(TUid,TUint,TInt,const TSecurityPolicy &,const TSecurityPolicy &,TInt);{}

RProperty::Define(TUint,TInt,const TSecurityPolicy &,const TSecurityPolicy &,TInt);{}

RSat::ClientSatProfileIndication(const TDesC8 &)const;{NetworkControl}

RSat::EventDownload(TRequestStatus &,TEventList,const TDesC8 &)const;{NetworkControl}

RSat::RefreshAllowed(TRequestStatus &,const TDesC8 &)const;{}

RSat::SendMessageNoLogging(TRequestStatus &,const TDesC8 &,TUint16 &)const;{NetworkServices}

RSoundPlugIn::Load(const TDesC &);{}

RSoundPlugIn::SetKeyClick(TBool);{}

RSoundPlugIn::SetPenClick(TBool);{}

RSoundPlugIn::Unload();{}

RSyncMLDataSyncProfile::CreateL(RSyncMLSession &);{}

RSyncMLDevMan::ClearRootAclL();{}

RSyncMLDevManProfile::CreateL(RSyncMLSession &);{}

RSyncMLHistoryLog::DeleteAllEntriesL();{}

RSyncMLSession::DeleteProfileL(TSmlProfileId);{}

RSyncMLSettings::SetValueL(TSmlGlobalSetting,TInt);{}

RWindowGroup::DefaultOwningWindow();{}

RWorldServer::DataFileRevertToSaved();{WriteUserData}

RWorldServer::DataFileSave();{WriteUserData}

RWorldServer::ResetAllData();{WriteUserData}

RWorldServer::SetHome(const TWorldId &);{WriteUserData}

RWorldServer::UpdateCity(TWorldId &,const TCityData &);{WriteUserData}

RWorldServer::UpdateCountry(TWorldId &,const TCountryData &);{WriteUserData}

RWsSession::ClaimSystemPointerCursorList();{}

RWsSession::SetClientCursorMode(TPointerCursorMode);{}

RWsSession::SetDefaultFadingParameters(TUint8,TUint8);{}

RWsSession::SetDoubleClick(const TTimeIntervalMicroSeconds32 &,TInt);{}

RWsSession::SetKeyboardRepeatRate(const TTimeIntervalMicroSeconds32 &,const TTimeIntervalMicroSeconds32 &);{}

RWsSession::SetModifierState(TEventModifier,TModifierState);{}

RWsSession::SetPointerCursorArea(TInt,const TRect &);{}

RWsSession::SetPointerCursorPosition(const TPoint &);{}

RWsSession::SetSystemFaded(TBool);{}

RWsSession::SetSystemFaded(TBool,TUint8,TUint8);{}

TExtendedLocale::SaveSystemSettings();{}

TExtendedLocale::SetCurrencySymbol(const TDesC &);{}

TLocale::Set()const;{}

User::SetCurrencySymbol(const TDesC &);{}

User::SetHomeTime(const TTime &);{}

User::SetMachineConfiguration(const TDesC8 &);{}

User::SetUTCOffset(TTimeIntervalSeconds);{}

User::SetUTCTime(const TTime &);{}

User::SetUTCTimeAndOffset(const TTime &,TTimeIntervalSeconds);{}

UserHal::RestoreXYInputCalibration(TDigitizerCalibrationType);{}

UserHal::SetXYInputCalibration(const TDigitizerCalibration &);{}

UserSvr::SetMemoryThresholds(TInt,TInt);{}

Capability: WriteUserData

CAgnAlarm::FindAndQueueNextAlarmL();{}

CAgnAlarm::RestartL();{}

CAgnAlarmActive::FindAndQueueNextAlarmL();{}

CAgnEntry::StoreNotesTextL();{}

CAgnEntry::UpdateNotesTextL();{ReadUserData}

CAgnEntryModel::AddCategoryToListL(const TDesC &);{}

CAgnEntryModel::AddEntryForServerL(CAgnEntry *,TAgnEntryId,TCommit,TUseExistingUniqueId);{}

CAgnEntryModel::AddEntryL(CAgnEntry *,TAgnEntryId);{ReadUserData}

CAgnEntryModel::AddEntryL(CAgnEntry *,TAgnEntryId,TCommit,TUseExistingUniqueId);{}

CAgnEntryModel::AddGsChildExceptionEntryL(CAgnEntry *,TAgnUniqueId,TGsBasicData,TTime);{}

CAgnEntryModel::AddGsChildRuleEntryL(CAgnEntry *,TAgnUniqueId,TGsBasicData,TTime);{}

CAgnEntryModel::AddGsEntryL(CAgnEntry *,HBufC8 *,TGsBasicData);{}

CAgnEntryModel::AddTodoListL(CAgnTodoList *,TInt);{ReadUserData}

CAgnEntryModel::ChangeTodoListOrderL(TInt,TInt);{ReadUserData}

CAgnEntryModel::ChangeTodoOrderL(CAgnTodoList *,TAgnEntryId,TAgnEntryId);{ReadUserData}

CAgnEntryModel::ChangeTodoOrderL(TAgnTodoListId,TAgnEntryId,TAgnEntryId);{ReadUserData}

CAgnEntryModel::CheckNotifier();{ReadUserData}

CAgnEntryModel::CreateL(RFs &,TFileName,const TDesC &,const CParaFormatLayer *,const CCharFormatLayer *);{}

CAgnEntryModel::CutEntryL(CAgnEntry *);{ReadUserData}

CAgnEntryModel::CutEntryL(TAgnEntryId);{ReadUserData}

CAgnEntryModel::DeleteAgendaFileL(const TDesC &)const;{}

CAgnEntryModel::DeleteEntryL(CAgnEntry *);{ReadUserData}

CAgnEntryModel::DeleteEntryL(TAgnEntryId);{ReadUserData}

CAgnEntryModel::DeleteEntryL(const HBufC8 &);{}

CAgnEntryModel::DeleteEntryL(const HBufC8 &,TTime);{}

CAgnEntryModel::DeleteTodoListL(CAgnTodoList *);{ReadUserData}

CAgnEntryModel::DeleteTodoListL(TAgnTodoListId);{ReadUserData}

CAgnEntryModel::DoChangeTodoOrderL(CAgnTodoList *,TAgnEntryId,TAgnEntryId);{ReadUserData}

CAgnEntryModel::DoSaveTodoListsL(CStreamStore &);{ReadUserData}

CAgnEntryModel::PasteEntryL(CAgnEntry *,TAgnEntryId);{ReadUserData}

CAgnEntryModel::QueueNextAlarmL(const TTime &,CArrayFixFlat< TAgnSortInstance > *);{}

CAgnEntryModel::RegisterObserverL(TUid,const TDesC8 &);{}

CAgnEntryModel::SecureSaveTodoListsL(const TDesC &)const;{}

CAgnEntryModel::SetParaAndCharFormatLayersL(const CParaFormatLayer *,const CCharFormatLayer *);{}

CAgnEntryModel::UnregisterObserverL(TUid);{}

CAgnEntryModel::UpdateEntryL(CAgnEntry *,TAgnEntryId);{ReadUserData}

CAgnEntryModel::UpdateObserverControllerL();{}

CAgnEntryModel::UpdateTodoListL(CAgnTodoList *);{ReadUserData}

CAgnIndexedModel::DeleteTidiedEntriesL();{ReadUserData}

CAgnIndexedModel::DoTidyByDateStepL();{ReadUserData}

CAgnIndexedModel::DoTidyByTodoListStepL();{ReadUserData}

CAgnIndexedModel::SetUpTidyByDateL(const TAgnFilter &,const TTime &,CStreamStore *,TStreamId &,TTidyDirective);{ReadUserData}

CAgnIndexedModel::SetUpTidyByTodoListL(CStreamStore *,TStreamId &,TTidyDirective);{ReadUserData}

CAgnIndexedModel::TidyByDateCompleted(TInt);{}

CAgnIndexedModel::TidyByDateL(const TAgnFilter &,const TTime &,const TTime &,const TTime &,MAgnProgressCallBack *,TTidyDirective);{ReadUserData}

CAgnIndexedModel::TidyByTodoListL(const CArrayFixFlat< TAgnTodoListId > *,MAgnProgressCallBack *,TTidyDirective,TTidyTodoListHow);{ReadUserData}

CAgnModel::AddEntryL(CAgnEntry *,TAgnEntryId);{}

CAgnModel::AddTodoListL(CAgnTodoList *,TInt);{ReadUserData}

CAgnModel::CutInstanceL(CAgnEntry *,TAgnWhichInstances);{ReadUserData}

CAgnModel::CutInstanceL(const TAgnInstanceId &,TAgnWhichInstances);{ReadUserData}

CAgnModel::DeleteInstanceL(CAgnEntry *,TAgnWhichInstances);{ReadUserData}

CAgnModel::DeleteInstanceL(const TAgnInstanceId &,TAgnWhichInstances);{}

CAgnModel::DeleteTodoListL(CAgnTodoList *);{ReadUserData}

CAgnModel::DeleteTodoListL(TAgnTodoListId);{ReadUserData}

CAgnModel::UpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{ReadUserData}

CAgnModel::UpdateTodoListL(CAgnTodoList *);{ReadUserData}

CCalCategoryManager::AddCategoryL(const CCalCategory &);{}

CCalDataExchange::ImportL(TUid,RReadStream &,RPointerArray< CCalEntry > &);{}

CCalDataExchange::ImportL(TUid,RReadStream &,RPointerArray< CCalEntry > &,TInt);{}

CCalEntryView::DeleteL(const CCalEntry &);{}

CCalEntryView::DeleteL(const CDesC8Array &);{}

CCalEntryView::StoreL(const RPointerArray< CCalEntry > &,TInt &);{}

CCalEntryView::UpdateL(const RPointerArray< CCalEntry > &,TInt &);{}

CCalInstanceView::DeleteL(CCalInstance *,CalCommon::TRecurrenceRange);{}

CCalSession::CreateCalFileL(const TDesC &)const;{}

CCalSession::DeleteCalFileL(const TDesC &)const;{}

CContactDatabase::CheckAndUpdatePhoneTableL();{}

CContactDatabase::CommitContactL(const CContactItem &);{ReadUserData}

CContactDatabase::CreateL(TThreadAccess);{}

CContactDatabase::CreateL(const TDesC &,TThreadAccess);{}

CContactDatabase::DatabaseBeginL(TBool);{}

CContactDatabase::DatabaseCommitL(TBool);{}

CContactDatabase::DatabaseRollback();{}

CContactDatabase::DeleteContactL(TContactItemId);{ReadUserData}

CContactDatabase::DeleteContactsL(const CContactIdArray &);{ReadUserData}

CContactDatabase::DeleteDatabaseL(const TDesC &);{}

CContactDatabase::DeleteDefaultFileL();{}

CContactDatabase::LockRecordLC(TContactItemId);{}

CContactDatabase::OpenContactL(TContactItemId);{}

CContactDatabase::OpenContactL(TContactItemId,const CContactItemViewDef &);{}

CContactDatabase::OpenContactLX(TContactItemId);{}

CContactDatabase::OpenContactLX(TContactItemId,const CContactItemViewDef &);{}

CContactDatabase::RecoverL();{}

CContactDatabase::RecreateSystemTemplateL(const TDesC &);{}

CContactDatabase::RemoveSpeedDialAttribsFromContactL(TContactItemId,TInt);{ReadUserData}

CContactDatabase::RemoveSpeedDialFieldL(TContactItemId,TInt);{ReadUserData}

CContactDatabase::ReplaceL(TThreadAccess);{}

CContactDatabase::ReplaceL(const TDesC &,TThreadAccess);{}

CContactDatabase::ResetServerSpeedDialsL();{}

CContactDatabase::SetCurrentDatabase(const TDesC &)const;{}

CContactDatabase::SetCurrentItem(const TContactItemId);{}

CContactDatabase::SetDatabaseDriveL(TDriveUnit,TBool);{}

CContactDatabase::SetFieldAsSpeedDialL(CContactItem &,TInt,TInt);{ReadUserData}

CContactDatabase::UpdateContactLC(TContactItemId,CContactItem *);{ReadUserData}

CContactDatabase::UpdateExistingContactL(CContactItem &);{}

CContactDatabase::doAddNewContactL(CContactItem &,TBool,TBool);{}

CContactDatabase::doCommitContactL(const CContactItem &,TBool,TBool);{ReadUserData}

CContactDatabase::doDeleteContactL(TContactItemId,TBool,TBool,TBool);{ReadUserData}

CContactDatabase::doDeleteContactsL(const CContactIdArray &,TBool &);{ReadUserData}

CContactDatabase::doOpenL(const TDesC &,TThreadAccess,TBool);{}

CFaxTransfer::AddSourceL(const TFileName &,TFaxPreferredCompression);{}

CFaxTransfer::AddSourceL(const TFileName &,TInt,TFaxPreferredCompression);{}

CFaxTransfer::AddSourceL(const TFileName &,TInt,TInt,TFaxPreferredCompression);{}

CFaxTransfer::RemoveAllSources();{}

CFaxTransfer::SetPhoneNumberL(TDesC8 &);{}

CFaxTransfer::Start(TRequestStatus &);{NetworkServices, ReadUserData}

CFaxTransfer::Stop();{NetworkServices, ReadUserData}

CFaxTransferSource::AddSourceL(const TFileName &,TFaxPreferredCompression);{}

CFaxTransferSource::AddSourceL(const TFileName &,TInt,TFaxPreferredCompression);{}

CFaxTransferSource::AddSourceL(const TFileName &,TInt,TInt,TFaxPreferredCompression);{}

CFaxTransferSource::RemoveAllSources();{}

CSmsEventLogger::AddEvent(TRequestStatus &,const CSmsMessage &,const TLogSmsPduData &,TInt *);{}

CSmsEventLogger::ChangeEvent(TRequestStatus &,const CSmsMessage &,const TLogSmsPduData &,TInt *);{}

CSmsEventLogger::DeleteEvent(TRequestStatus &);{}

CSmsHeader::ExternalizeL(RMsvWriteStream &)const;{}

CSmsHeader::StoreL(CMsvStore &)const;{}

CUnifiedCertStore::Remove(const CCTCertInfo &,TRequestStatus &);{WriteDeviceData}

CUnifiedKeyStore::CreateKey(TInt,TKeyUsagePKCS15,TUint,const TDesC &,CCTKeyInfo::EKeyAlgorithm,TInt,TTime,TTime,CCTKeyInfo *&,TRequestStatus &);{}

CUnifiedKeyStore::DeleteKey(TCTTokenObjectHandle,TRequestStatus &);{}

CUnifiedKeyStore::ImportKey(TInt,const TDesC8 &,TKeyUsagePKCS15,const TDesC &,TInt,TTime,TTime,CCTKeyInfo *&,TRequestStatus &);{}

CUnifiedKeyStore::SetManagementPolicy(TCTTokenObjectHandle,const TSecurityPolicy &,TRequestStatus &);{}

CUnifiedKeyStore::SetPassphraseTimeout(TInt,TRequestStatus &);{}

CUnifiedKeyStore::SetUsePolicy(TCTTokenObjectHandle,const TSecurityPolicy &,TRequestStatus &);{}

MCTKeyStoreManager::CreateKey(CCTKeyInfo *&,TRequestStatus &)=0;{}

MCTKeyStoreManager::ImportEncryptedKey(const TDesC8 &,CCTKeyInfo *&,TRequestStatus &)=0;{}

MCTKeyStoreManager::ImportKey(const TDesC8 &,CCTKeyInfo *&,TRequestStatus &)=0;{}

MCTWritableCertStore::Add(const TDesC &,TCertificateFormat,TCertificateOwnerType,const TKeyIdentifier *,const TKeyIdentifier *,const TDesC8 &,TRequestStatus &)=0;{WriteDeviceData}

MCTWritableCertStore::Add(const TDesC &,TCertificateFormat,TCertificateOwnerType,const TKeyIdentifier *,const TKeyIdentifier *,const TDesC8 &,const TBool,TRequestStatus &);{WriteDeviceData}

MCTWritableCertStore::Remove(const CCTCertInfo &,TRequestStatus &)=0;{WriteDeviceData}

RAgendaServ::AddCategoryToListL(const TDesC &);{}

RAgendaServ::AddEntryL(CAgnEntry *,TAgnEntryId,TInt,TBool);{ReadUserData}

RAgendaServ::AddTodoListL(CAgnTodoList *,TInt);{}

RAgendaServ::ChangeTodoListOrderL(TInt,TInt);{}

RAgendaServ::CloseWriteStreamL();{}

RAgendaServ::CreateNewStreamL(TUid);{}

RAgendaServ::DecEntryRefCountL(const TAgnUniqueId);{}

RAgendaServ::DecEntryRefCountsL(const CArrayFix< TAgnUniqueId > &);{}

RAgendaServ::DeleteEntry(TAgnUniqueId);{}

RAgendaServ::DeleteEntryL(CAgnEntry *);{ReadUserData}

RAgendaServ::DeleteNotesTextL(TStreamId);{}

RAgendaServ::DeleteTodoList(TAgnUniqueId);{}

RAgendaServ::DeleteTodoListL(CAgnTodoList *);{}

RAgendaServ::IncEntryRefCountL(const TAgnUniqueId);{}

RAgendaServ::IncEntryRefCountsL(const CArrayFix< TAgnUniqueId > &);{}

RAgendaServ::QueueNextAlarmL(const TTime &,CArrayFixFlat< TAgnSortInstance > *,CAgnSortEntryAllocator *);{}

RAgendaServ::SaveTodoListsL(TFileName);{}

RAgendaServ::SetDefaultDisplayTimes(TTimeIntervalMinutes,TTimeIntervalMinutes,TTimeIntervalMinutes);{}

RAgendaServ::SetParaAndCharFormatLayersL(const CParaFormatLayer *,const CCharFormatLayer *);{}

RAgendaServ::StoreNotesTextL(const TDesC &);{}

RAgendaServ::UpdateEntryL(CAgnEntry *,TAgnEntryId);{ReadUserData}

RAgendaServ::UpdateNotesTextL(const TDesC &,TStreamId);{}

RAgendaServ::UpdateTodoListL(CAgnTodoList *);{}

RFax::Write(TRequestStatus &,const TDesC8 &);{NetworkServices}

RMobileBroadcastMessaging::SetFilterSetting(TRequestStatus &,TMobilePhoneBroadcastFilter)const;{}

RMobileBroadcastMessaging::SetLanguageFilter(TRequestStatus &,const TDesC16 &)const;{}

RMobileBroadcastMessaging::StoreBroadcastIdListL(TRequestStatus &,CMobilePhoneBroadcastIdList *,TMobileBroadcastIdType);{}

RMobileCall::AnswerIncomingCallWithUUI(TRequestStatus &,const TDesC8 &,const TMobileCallUUI &)const;{NetworkServices}

RMobileCall::HangupWithUUI(TRequestStatus &,const TMobileCallUUI &)const;{NetworkServices}

RMobileCall::SendUUI(TRequestStatus &,TBool,const TMobileCallUUI &)const;{NetworkServices}

RMobileONStore::StoreAllL(TRequestStatus &,CMobilePhoneONList *)const;{}

RMobilePhoneBookStore::Write(TRequestStatus &,const TDesC8 &,TInt &)const;{}

RMobilePhoneStore::Delete(TRequestStatus &,TInt)const;{}

RMobilePhoneStore::DeleteAll(TRequestStatus &)const;{}

RMobilePhoneStore::Write(TRequestStatus &,TDes8 &)const;{}

RMobileSmsMessaging::SendMessage(TRequestStatus &,const TDesC8 &,TDes8 &)const;{NetworkServices}

RMsvServerSession::ChangeAttributesL(const CMsvEntrySelection &,TUint,TUint);{}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::ChangeEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::CopyEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp);{None, ReadUserData}

RMsvServerSession::CopyEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp,TRequestStatus &);{None, ReadUserData}

RMsvServerSession::CreateAttachmentForWriteL(TMsvId,TDes &,RFile &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::CreateEntryL(const TMsvEntry &,TMsvOp,TSecureId,TRequestStatus &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::DeleteAttachment(TMsvId,const TDesC &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::DeleteEntriesL(const CMsvEntrySelection &,TMsvOp,TRequestStatus &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::DeleteFileStoreL(TMsvId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::LockStore(TMsvId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::MoveEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp);{None, ReadUserData}

RMsvServerSession::MoveEntriesL(const CMsvEntrySelection &,TMsvId,TMsvOp,TRequestStatus &);{None, ReadUserData}

RMsvServerSession::OpenAttachmentForWriteL(TMsvId,const TDesC &,RFile &);{None, ReadUserData}

RMsvServerSession::OpenTempStoreFileL(TMsvId,RFile &);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::ReleaseStore(TMsvId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::RemoveEntry(TMsvId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::ReplaceFileStoreL(TMsvId);{None, ReadUserData, WriteDeviceData}

RMsvServerSession::TransferCommandL(const CMsvEntrySelection &,TInt,const TDesC8 &,TMsvOp);{ReadUserData}

RMsvServerSession::TransferCommandL(const CMsvEntrySelection &,TInt,const TDesC8 &,TMsvOp,TRequestStatus &);{ReadUserData}

RPhoneBookSession::CancelRequest(TPhonebookSyncRequestCancel,TUid);{ReadUserData}

RPhoneBookSession::DeleteContact(TRequestStatus &,TContactItemId);{}

RPhoneBookSession::DoSynchronisation(TRequestStatus &);{ReadUserData}

RPhoneBookSession::DoSynchronisation(TRequestStatus &,TUid);{ReadUserData}

RPhoneBookSession::ShutdownServer(TBool);{ReadUserData}

RPhoneBookSession::WriteContact(TRequestStatus &,CContactICCEntry &,TInt &);{}

RPhoneBookSession::WriteContact(TRequestStatus &,CContactICCEntry &,TInt &,TUid &);{}

RWorldServer::DataFileRevertToSaved();{WriteDeviceData}

RWorldServer::DataFileSave();{WriteDeviceData}

RWorldServer::ResetAllData();{WriteDeviceData}

RWorldServer::SetHome(const TWorldId &);{WriteDeviceData}

RWorldServer::SetRomDatabaseToUse(const TDesC &);{}

RWorldServer::UpdateCity(TWorldId &,const TCityData &);{WriteDeviceData}

RWorldServer::UpdateCountry(TWorldId &,const TCountryData &);{WriteDeviceData}

TAgnInstanceEditor::CreateAndStoreExceptionL(CAgnEntry *&,CAgnEntry *,TAgnEntryId);{ReadUserData}

TAgnInstanceEditor::DeleteInstanceL(CAgnEntry *,TAgnWhichInstances);{ReadUserData}

TAgnInstanceEditor::DoUpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{ReadUserData}

TAgnInstanceEditor::SplitRepeatL(CAgnEntry *&,TAgnWhichInstances,CAgnEntry *,TAgnEntryId);{ReadUserData}

TAgnInstanceEditor::UpdateInstanceL(CAgnEntry *,TAgnWhichInstances,TAgnEntryId);{ReadUserData}

Capability: Illegal

MCTDH::Agree(const CDHPublicKey &,HBufC8 *&,TRequestStatus &)=0;{}

MCTDH::PublicKey(const TInteger &,const TInteger &,CDHPublicKey *&,TRequestStatus &)=0;{}

MCTDecryptor::Decrypt(const TDesC8 &,TDes8 &,TRequestStatus &)=0;{}

MCTKeyStoreManager::DeleteKey(TCTTokenObjectHandle,TRequestStatus &)=0;{}

MCTKeyStoreManager::ExportEncryptedKey(TCTTokenObjectHandle,const CPBEncryptParms &,HBufC8 *&,TRequestStatus &)=0;{}

MCTKeyStoreManager::ExportKey(TCTTokenObjectHandle,HBufC8 *&,TRequestStatus &)=0;{}

MCTKeyStoreManager::SetManagementPolicy(TCTTokenObjectHandle,const TSecurityPolicy &,TRequestStatus &)=0;{}

MCTKeyStoreManager::SetPassphraseTimeout(TInt,TRequestStatus &)=0;{}

MCTKeyStoreManager::SetUsePolicy(TCTTokenObjectHandle,const TSecurityPolicy &,TRequestStatus &)=0;{}

MCTSigner::Sign(const TDesC8 &,Signature &,TRequestStatus &)=0;{}

MCTSigner::SignMessage(const TDesC8 &,Signature &,TRequestStatus &)=0;{}

MKeyStore::ExportPublic(const TCTTokenObjectHandle &,HBufC8 *&,TRequestStatus &)=0;{}

MKeyStore::GetKeyInfo(TCTTokenObjectHandle,CCTKeyInfo *&,TRequestStatus &)=0;{}

MKeyStore::Open(const TCTTokenObjectHandle &,MCTDH *&,TRequestStatus &)=0;{}

MKeyStore::Open(const TCTTokenObjectHandle &,MCTDecryptor *&,TRequestStatus &)=0;{}

MKeyStore::Open(const TCTTokenObjectHandle &,MDSASigner *&,TRequestStatus &)=0;{}

MKeyStore::Open(const TCTTokenObjectHandle &,MRSASigner *&,TRequestStatus &)=0;{}

RAvctp::Open(MAvctpEventNotify &,SymbianAvctp::TPid);{}

RComm::Open(RCommServ &,const TDesC &,TCommAccess);{}

RComm::Open(RCommServ &,const TDesC &,TCommAccess,TCommRole);{}

RComm::OpenWhenAvailable(TRequestStatus &,RCommServ &,const TDesC &);{}

RComm::OpenWhenAvailable(TRequestStatus &,RCommServ &,const TDesC &,TCommRole);{}

RFs::Entry(const TDesC &,TEntry &)const;{}

RSocket::Open(RSocketServ &,TUint,TUint,TUint);{}

RSocket::SetOpt(TUint,TUint,const TDesC8 &);{Dependent}

RTz::SetTimeZoneL(CTzId &)const;{}