QUALCOMM®

Binary Runtime Environment for Wireless®

# BREW® 3.1.2 OEM Porting Guide for MSM Platforms

brew™

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA. 92121-1714
U.S.A.

BREW® 3.1.2 OEM Porting Guide for MSM Platforms
November 18, 2004

# Contents

# Introducing the BREW OEM Porting Guide for MSM Platforms

This document provides information about and instructions for porting QUALCOMM's Binary Runtime Environment for Wireless® (BREW™) platform to mobile devices that use the Mobile Station Modem (MSM™) family of ASICs from QUALCOMM CDMA Technologies (QCT).

## What's in this guide

The remainder of the *BREW™ OEM Porting Guide* contains the following sections.

| | |
|---|---|
| OEM Porting Kit Overview | Provides a general overview of the porting process, and describes the components of the Porting Kit and their relationships to one another. |
| Quick-Start Guide to Porting | Contains a high-level tutorial that results in installing a simple application on a device, and gives references to subsequent sections containing information specific to certain Porting Kit components and porting tasks. |
| Initializing BREW | Provides information about plugging BREW into a task that enables it to communicate with the native device code, and tells you how to invoke the initialization routine in the device code to initialize BREW. |
| Sending Events to BREW | Discusses event handling. |
| Implementing Display Support | Provides specifics about displaying BREW applications on devices, including using the BREW Simulator. |
| Understanding the Generic Serial Interface | Explains the BREW Serial I/O (SIO). |
| Configuring Devices | Explains the various aspects of device configuration for BREW. |

| | |
|---|---|
| Managing and Downloading Applications and Extensions | Discusses application download capabilities including dynamic and static application extensions. |
| BREW UI Guidelines | Explains the various aspects of the BREW UI and discusses best practices for integrating BREW with your native UI. |
| Memory Security Through BRIDLE | Describes the BREW Isolated Domain for Legitimate Execution (BRIDLE) mechanism used to protect BREW and Dual Mode Subscriber Software (DMSS) code and data from BREW applications. |
| Setting Up Call Handling | Explains how to use the ITAPI interface for call handling. |
| Setting up SMS | Explains how to set up and verify the implementation of SMS interfaces. |
| Setting up battery | Explains how to set up and verify the implementation of the IBattery interface. |
| Managing Resources | Describes how BREW manages resources using the resource controller and resource arbiter. |
| Interoperability with GSM1x | Explains interoperability between CDMA and GSM networks. |
| OEM Acceptance Process | Discusses the integration and acceptance test process. |
| Static Modules | Describes version control of static applications and modules through the use of Module Information Files (MIF). |
| Appendix A: Using the OEM Extranet | Provides details for obtaining and using a BREW OEM Extranet account. |
| Appendix B: DMI Compliance | Provides information regarding the Diagnostic Monitor Interface (DMI). |
| Appendix C: Test Enable Bit Removal | Includes some conceptual information about the removal of the test enabled bit and a table showing functionality removed and the current method to support this functionality. |

| | |
|---|---|
| Appendix C: Test Enable Bit Removal | Includes some conceptual information about the removal of the test enabled bit and a table showing functionality removed and the current method to support this functionality. |

For information about specific interfaces, please see the *BREW OEM API Reference Online Help*.

## Acronyms and terms

The following acronyms and terms are used throughout the Porting Kit documentation set.

| | | |
|---|---|---|
| ADS | Application Download Server | The ADS hosts the carrier's catalog of BREW applications and is the host device to which subscribers connect for catalog browsing and application downloads. |
| AMSS | Advanced Mode Subscriber Software | |
| APCS | ARM Procedure Call Standard | |
| ATCOP | AT Command Processor | |
| BAR | BREW Applet Resource file | The binary output file from the Resource Editor. |
| BBF | BREW Bitmapped Font | BREW-specific font format. |
| BCI | BREW Compressed Image | A BCI file consists of a series of graphic images compressed and combined, using the BREW Compressed Image Authoring Tool, to add animation to a BREW application. |
| BRIDLE | BREW Isolated Domain for Legitimate Execution | A mechanism to protect BREW and DMSS code and data from BREW applications. The mechanism uses the services of the Memory Management Unit (MMU) available on MSM™ chipsets with ARM 9 core. In the absence of the MMU, as in the case of ARM7TDMI based MSM ASICS, this mechanism can be used to prevent BREW applications from disabling interrupts, modifying call stacks, and the like. |
| CAVE | Cellular Authentication and Voice Encryption | An algorithm used by mobiles devices for authentication with the base station. |
| CHV | Card holder verification | |
| DDB | Device dependent bitmap | Bitmap in the device's native format. |
| DIB | Device independent bitmap | An object that supports the IBitmap or IDIB interfaces. |
| DMSS | Dual Mode Subscriber Software | |
| DMI | Diagnostic Monitor Interface | A QCT (DMSS) interface used by all BREW tools, including the BREW AppLoader, BREW Logger, The Grinder®, and Shaker. |
| DSP | Digital signal processor | A specialized computer chip designed to perform speedy and complex operations on digitized waveforms. |

| | | |
|---|---|---|
| DTR | Data terminal ready | A control signal sent from the Data Terminal Equivalent (DTE) to the Data Communications Equivalent (DCE) that indicates that the DTE is powered on and ready to communicate. DTR can also be used for hardware flow control. |
| EF | Elementary file | |
| EULA | End User License Agreement | An agreement requested during installation sequences that binds the user to copyright responsibilities. |
| ICE | In-circuit emulator | |
| ISOD | Interface Specification and Operational Description | A QCT guide that explains how to use interfaces for services provided by the modem baseband processor. |
| JTAG | Joint test action group | |
| MD5 | Message-Digest 5 | An RSA hash algorithm developed by Ronald L. Rivest of MIT to verify data integrity. |
| ME | Mobile equipment | |
| MIF | Module Information File | The MIF Editor generates this binary file, which contains information regarding the list of classes and applets supported by the modules. |
| MMU | Memory Management Unit | The platforms that contains MMU with ARM9 chipsets, for example: MSM6100 |
| MOD | Dynamically loaded module | This file type is the dynamically loaded module executed at runtime. The applet source files are compiled and linked into this MOD file type. |
| MO-SMS | Mobile originated SMS | A method of sending short alphanumeric messages from a mobile device. In BREW, the OEM is responsible for implementing MO-SMS on the OEM code layer to allow BREW and BREW applications to send SMS. |
| MPU | Memory Protection Unit | The platforms that contain the MMU with ARM7 based chipsets, for example, MSM6050™. |
| OAT | Operational Acceptance Test | A PEK tool that provides a set of tests that allow verification of BREW porting on a device. |
| OTA | Over-the-air | |
| PEK | Porting Evaluation Kit | A QIS product that tests the operational accuracy of BREW porting and BREW performance on a device. |
| PNG | Portable Network Graphics Format | A graphics file format that uses advanced image compression technology to provide better color depth for 16-, 24-, and 32-bit images. |
| PRL | Preferred roaming list | |
| REX | Qualcomm’s real-time executive operating system | |
| RLP | Radio link protocol | |

| | | |
|---|---|---|
| R-UIM | Removable user identity module | A collection of functions that verifies the R-UIM connection and returns the R-UIM status on a R-UIM-based device. |
| SID | Subscriber ID | Identifies the device involved in a BREW application transaction. |
| SIG | Signature file | A file attached to dynamic applications for verification purposes. |
| SIO | Serial input/output | The electronic methodology used in serial data transmission. |
| SPC | Service programming code | A device-specific code required to install BREW on a device. |
| SWI | Software interrupt | |
| UART | Universal asynchronous receiver/transmitter | A device, usually an integrated circuit chip, that performs the parallel-to-serial conversion of digital data to be transmitted and the serial-to-parallel conversion of digital data that has been transmitted. |
| UASMS | UI API for SMS | A function for registering SMS listener functions in BREW. |
| UIMID | User Identity Module ID | |

# OEM Porting Kit Overview

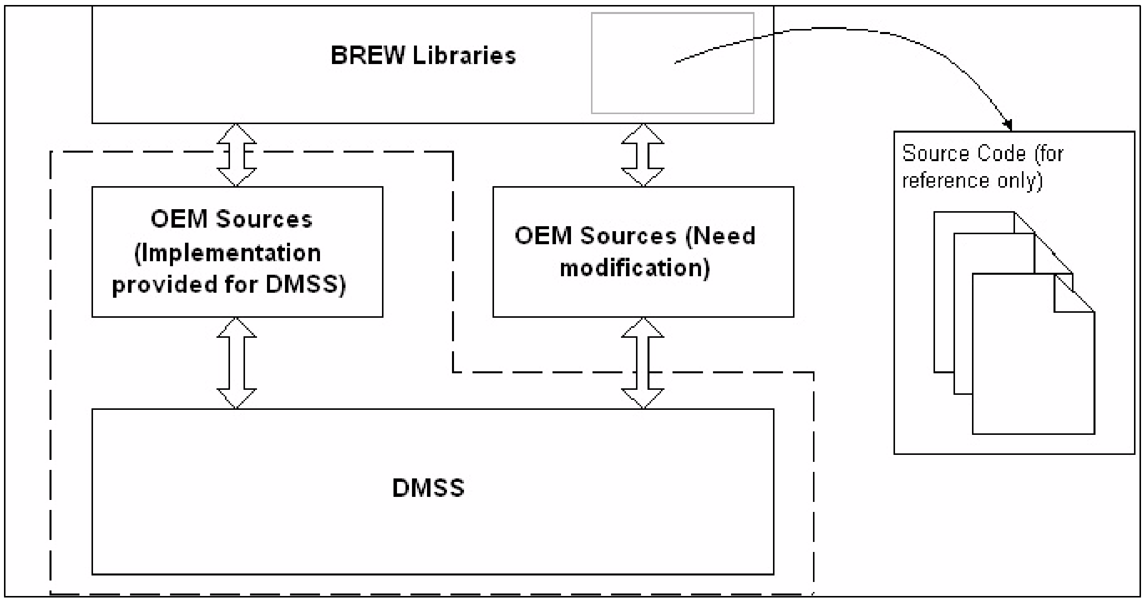
BREW provides a layer on top of the chip system software. This makes the device's functionality available to the application without requiring the developer to have the chip system source code or even a direct relationship with the device manufacturer.

This section describes the components of the Porting Kit and their relationships to one another.

## What's in the Porting Kit?

The diagram below shows the components that comprise the OEM Porting Kit. A table of component descriptions follows the diagram.

*OEM Porting Kit components*


BREW Libraries
OEM Sources (Implementation provided for DMSS)
OEM Sources (Need modification)
Source Code (for reference only)
DMSS

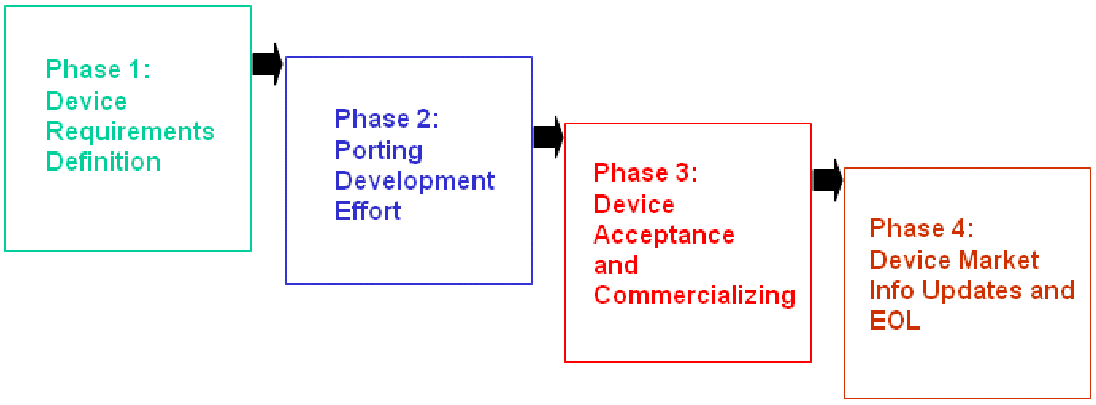
## *Porting Kit Components*

| Component | Description |
|---|---|
| BREW library files (BREW*.lib) | The library files are available for linking with the device software using the ARM Developer Suite.<br>For information about using the libraries and ARM, see the *BREW OEM API Reference Online Help*. |
| OEM reference implementation files | The OEM reference implementation files are compiled and linked with the BREW libraries and the device build. OEM interfaces must be integrated to support applications using BREW. The reference implementations for the non-device driver specific OEM interfaces (for example, LCD) are provided to you as part of the Porting Kit. For information about specific OEM interface files, refer to the *BREW™ OEM API Reference Online Help for MSM Platforms* or the OEM source file headers. |
| AEE header files (AEE*.h) | Referenced for the device build, the AEE header files expose the BREW API to BREW applications as well as to the native applications that use the functions exposed by the API. These files are described in detail in the *BREW OEM API Reference Online Help*. |
| AEE source files | The AEE source files correspond to some of the BREW library files, and are already built into the BREW libraries. The main purpose of providing these files in source is to assist you with debugging, if it is necessary. It is recommended that you do not modify these files, as this will likely cause instability or incompatibility with future software versions and failure of applications that leverage these interfaces. To use any of these file for debugging purposes, compile the files into their corresponding object files (.o files). Place the .o files ahead of the BREW libraries when the link command is issued to build the device image. This forces the linker to use the symbols in the .o files, not those in the BREW libraries. |

## BREW from an integration perspective

The following diagram shows the BREW device phases. The remainder of this document addresses Phase 2, the Porting Development Effort

***BREW device phases.***


Phase 1: Device Requirements Definition
Phase 2: Porting Development Effort
Phase 3: Device Acceptance and Commercializing
Phase 4: Device Market Info Updates and EOL

The following is a description of the phases:

| | |
|---|---|
| Phase 1 | The operator communicates its device requirements to the OEM. Phase 1 concludes when the operator and OEM reach an agreement on the features of the BREW device. This agreed-upon features list becomes a formal requirements document. |
| Phase 2 | The OEM is involved in porting BREW onto the device. At the end of phase 2, the OEM has completed the port and has run the PEK to verify that the port meets the requirements specified in Phase 1. |
| Phase 3 | The operator is responsible for testing the device. If necessary, QIS performs a readiness review of the device to determine if application testing can be commenced on that device. At the end of phase 3, the commercial viability of the device is determined. |
| Phase 4 | Developers are provided with any device software updates. Also, the operator determines if the device has reached end of life, in which case no more applications are accepted for that device. |

# Quick-Start Guide to Porting

This section contains procedures that will give you hands-on practice to familiarize you with the main aspects of the porting process. Where appropriate, procedural steps are accompanied by references to sections containing more detailed information.

## Before you begin

The following hardware tools and software are required for porting:

- ARM Development Suite 1.0, 1.1, 1.2 or equivalent compiler/linker with the latest patches
- JTAG or ICE debugging device and software
- Data cable to connect to the serial port of the device
- Product Support Tools (PST) or equivalent tools to access:
  - File system
  - Non-volatile items or other non-volatile configuration items
  - Image download
  - Debug messages

If you intend to develop custom extensions or applications utilizing the device simulator included in the Porting Kit, it is recommended that you also have Microsoft Visual C++ 6.0 or higher.

## Step 1. Initialize BREW

1. Identify the DMSS task in whose context BREW will run. This may be an existing task, such as the User Interface task, or a new task created specifically for BREW. The stack of the task in which BREW is integrated is used by BREW and all of the applications. It is recommended to keep this size as high as possible. Typical values are 8 - 16 KB.

**NOTE:** *BREW is not re-entrant.* Accordingly, all calls to BREW APIs must be made in the context of the task in which BREW was initialized.

2. Define a new and unique signal AEE_APP_SIG for BREW in the task identified in 1, above.
3. Modify the task loop for the BREW task identified, to execute AEE_Dispatch() if AEE_APP_SIG is set.
4. Initialize the BRIDLE SWI Handler table by calling bridle_InitSWITable(). This function must be called before initializing BREW, regardless of whether BRIDLE is turned on.
5. Add code to initialize BREW by calling AEE_Init(AEE_APP_SIG). This must be done before any BREW service or interface is invoked. It is typically done in the task initialization code for the BREW task.
6. Add code to terminate BREW by calling AEE_Exit(). AEE_Exit() must be called before the device is powered down, to allow BREW to properly perform its shutdown procedures.

See Initializing BREW on page 23.

## Step 2. Set up event handling

1. Translate each key input into an AEE Virtual Key code, and send an event by means of each of the following functions:
   - AEE_Key(wVCode)
   - AEE_KeyPress (wVCode)
   - AEE_KeyRelease(wVCode)

   **NOTE:** These functions must be called in the context of the task in which BREW was initialized.

   You must send key events (EVT_KEY_PRESS, EVT_KEY, EVT_KEY) to BREW for each key pressed and released. For the complete list of the AEE Virtual Key codes, see the definition of AVKType in the *BREW SDK API Reference Online Help*.

2. Send the following events to BREW:

   - AEE_SEND_FLIP_EVT (p)
   - AEE_SEND_KEYGUARD_EVT (p)
   - AEE_SEND_LOCKED_EVT (p)
   - AEE_SEND_HEADSET_EVT (p)
   - AEE_SEND_SCRROTATE_EVT (p)

   **NOTE:** All key and system events must be sent to BREW, regardless of whether BREW is suspended or there is an active BREW application. These events reset the BREW screen saver inactivity timer.

See Sending Events to BREW on page 27.

## Step 3. Implement display support

1. Choose an OEMBitmap implementation that best matches your screen color bit depth:

   - 1 bit per pixel, 2 colors (usually black and white), OEMBitmap1.c
   - 2 bits per pixel, 4 colors (usually grey scale), OEMBitmap2.c
   - 8 bits per pixel, 256 colors, OEMBitmap8.c
   - 12 bits per pixel, 4096 colors, OEMBitmap12.c
   - 15/16 bits per pixel, 65536 colors, OEMBitmap16.c
   - 18 bits per pixel, 262144 colors, OEMBitmap18.c

   The color bit-depth does not refer to the physical LCD capability; you should pick one that matches the color depth of your screen buffer.

2. Using OEMDisplayDev.c as a template, implement the IDisplayDev interface for the device.

3. Implement the AEECLSID_DEVBITMAPn and AEECLSID_DEVBITMAPn_CHILD ClassIDs for this display to link to one of the bitmap implementations chosen in step 1.

   **NOTE:** Be sure that the pixel buffer provided to the bitmap constructor is large enough to hold the bitmap data for any of the bitmap formats selected in step 1. For instance, if the OEMBitmap16 and OEMBitmap18 implementations have been selected, the buffer must be big enough to hold the pixel data for the OEMBitmap18 implementation.

4. Using OEMAppFrame.c as an example, implement the IAppFrame interface for this display. Link it into the OEM mod table as demonstrated in the reference implementation using the appropriate ClassID (AEECLSID_APPFRAMEn, where n is the display number, 1-4).
5. Implement the AEE_DEVICEITEM_SYS_COLORS_DISPn and AEE_DEVICEITEM_DISPINFOn device items in OEM_GetDeviceInfoEx() (OEMConfig.c), where n is the display number.
6. This step is for the primary display, only:
   Modify OEM_GetDeviceInfo() to put the appropriated values for the primary display in the cxScreen, cyScreen, and nColorDepth fields of the AEEDeviceInfo struct. (cxAltScreen and cyAltScreen are deprecated and should be set to 0.)
7. Enable individual annunciators, depending on carrier requirements, by implementing OEM_Disp_Annunciators. See the BREW OEM API Reference Online Help.

Refer to Implementing Display Support on page 36.

## Step 4. Configure the device

1. Provide device information to BREW (OEM_GetDeviceInfo(), OEM_GetDeviceInfoEx()).
2. Provide configuration information to BREW (OEM_GetConfig()).
3. Provide mapping for the directory names between BREW and native device code. The mapping is defined by OEMFSGNPMap structure for the following name spaces:
   - AEEFS_ROOT_DIR
   - AEEFS_HOME_DIR
   - AEEFS_SYS_DIR
   - AEEFS_MOD_DIR
   - AEEFS_MIF_DIR
   - AEEFS_SHARED_DIR
   - AEEFS_ADDRESS_DIR
   - AEEFS_RINGERS_DIR
   - AEEFS_CARD0_DIR

Refer to Configuring Devices on page 63.

## Step 5. Perform the first device build

1. Verify the existing version of BREW that is on the DMSS/AMSS software by checking the BREWVersion.h file located in apps/BREW/inc.
2. If you already have a build with BREW version 3.0 on it, you can just build the standard build with no modifications.
3. If you have a build with BREW version 2.1, you must make some modifications to the build system and to the DMSS/AMSS software. Detailed instructions are located in *BREW OEM Note: Migrating From BREW 2.1 to 3.0*. The basic steps required are:
   - **a.** Make sure all BREW libraries are included in the build. (For information about BREW libraries, see the *BREW OEM API Reference Online Help*.)
   - **b.** Compile all of the needed OEM files.
   - **c.** Make sure all static apps have MIF files, and that these are included in OEMConstFiles.c
   - **d.** Reconcile changes from BREW 2.1 to BREW 3.0 with the DMSS/AMSS software.

## Step 6. Enable BREW features

The list of required BREW features varies from operator to operator. Refer to the Enabling and Testing Instructions for BREW Interfaces MSM for a description of all BREW features, along with specific instructions for enabling each required feature. The enabling and testing instructions also includes details such as whether a reference implementation is provided for a feature, how to replace the reference implementation with your own implementation, and which BREW libraries support the feature (if any).

## Step 7. Enable the BREW Application Manager and OTA downloads

1. Enable BREWAppManager.

a. Link the appropriate BrewAppMgr library into the device build.

b. Copy the appropriate color depth brewappmgr.mif in .../brew/mifs folder.

c. Copy the appropriate color depth brewappmgr.bar in .../brew/mods/brewappmgr folder.

Refer to Enabling the BREW Application Manager for OTA download on page 90.

## Step 8. Integrate the native UI within BREW-enabled devices

1. Integrate shim application-related code with your project. You can copy the code from the <BREW\pk\examples\shimapphelper> directory included with the Porting Kit.
2. Identify all native applications to be shimmed, #define the quantity, and create a list of their unique class Ids.
3. Add an application for each native application to the OEMTransientApp. These applications will all behave the same by default.
4. Add the transient and idle applications to the mod table, so that BREW can access them.
5. Locate your common event handler function, define a function pointer to describe it, map each event handler function to the class ID associated with the application, and create a map lookup table.
6. Add the Idle application as the system's auto-start application.
7. Edit your application state mechanism to launch and close shimmed applications, rather than pushing or popping a new major state onto the state machine's stack. This ensures that the BREW state remains running, and your native application runs under the BREW shim created earlier.
8. Look up and invoke the event handler for the application, as needed. This is determined by whether a shim application is running. The event handler must be invoked when the application is started, or when a device event occurs.
9. Enable/disable Privacy Check in ITAPI_MakeVoiceCall() and IPOSDET_GetGPSInfo(). Refer to Privacy check removal on page 128.

Refer to BREW UI Guidelines on page 103.

## Step 9. Protect memory with BRIDLE

1. Port BREW (with BRIDLE turned off). Ensure that bridle_InitSWITable is called before AEE_Init.
2. Create code and data regions as described in the scatter load section. Modify the boot loader to use the new load and exec regions (bootmem.c).
3. Replace the SWI Handler in DMSS with the SWI Handler provided in the Porting Kit (swihandler.s)
4. Configure the MMU/MPU
5. Incorporate any changes to DMSS as recommended in the BREW release notes.
6. Enable MMU/MPU and set bridle_on to TRUE.

Refer to  Memory Security Through BRIDLE on page 94.

## Step 10. Implement the Resource Manager

1. Enable IResourceControl. This enables IResourceControl classes for application priority (AEECLSID_TOPVISIBLECTL) and sound (AEECLSID_RESCTL_SOUND).
2. Build IResArbiter class implemented in OEMResArbiter.c. You can customize the resource arbiter logic based on your requirements.

## Step 11. Perform testing

3. Working with the operator, complete the BREW Device Requirements questionnaire.
4. Complete a Device Data Form/Device Pack using the BRWE Device Configurator.
5. Run and pass the tests in the BREW Porting Evaluation Kit (PEK).
6. Test the end-user experience.

See OEM Acceptance Process on page 154, and the *BREW Porting Evaluation Kit (PEK) User's Guide*.

# Initializing BREW

This section discusses pre-initialization procedures and describes the BREW initialization process.

## Getting ready to initialize

Before BREW can be initialized, you must identify a task in which to run BREW, define a unique task signal, modify the task loop, and initialize the BRIDLE SWI Handler.

### Identifying the BREW task

You must identify a task in which to run BREW, and then define a unique task signal so that BREW can communicate with the native device code. You can identify either a new task or an existing task; however, QUALCOMM strongly recommends that you use the UI task for the BREW task. In the event you need to create a separate task for BREW or run on another preexisting task, the priority of the task must be immediately under the UI task priority, but higher than the file system and sleep task priorities.

**TIP:** Numerous calls are made in the BREW task to display-related routines (for example, DrawText() and Drawing Images). If the drawing routines do not already run in the BREW task, be sure that those routines can handle being called from multiple threads.

**NOTE:** For DMSS releases that already include BREW, BREW runs in the UI task. Reference task.h to locate the priority of the UI task.

### Defining a task signal

After a task has been identified, you must define a unique task signal so BREW can communicate with the native device code. BREW sets the AEE_APP_SIG signal to inform the native device code that BREW needs to execute.

For example, for devices using MSM chips and running BREW in the UI task context, add a signal AEE_APP_SIG in ui.h for BREW. This signal must be unique to the BREW task and must not conflict with any other general-purpose REX signal.

The following code shows a common way of adding a handler for the AEE_APP_SIG signal in the task function, which is responsible for running BREW:

```
static rex_sigs_type sigs;    // hold signals
...
for( ; ; )
{
    sigs = rex_wait(
    ...
           | AEE_APP_SIG);      // wait for AEE_APP_SIG
    ...
    if( sigs & AEE_APP_SIG ) { // handle AEE_APP_SIG
      (void) rex_clr_sigs( &ui_tcb, AEE_APP_SIG);
      AEE_Dispatch();
    }
    ...
}
```

**NOTE:** Do not put any conditional statements around the AEE_Dispatch() call. Whenever the task running BREW receives the BREW signal, AEE_Dispatch() must be called.

**NOTE:** For DMSS releases that already include BREW, find the definition of AEE_APP_SIG in ui.h. The ui_task() function in ui.c call rex_wait() for AEE_APP_SIG. When rex_wait returns, the signals are passed to ui_signal() in uihsig.c, which calls AEE_Dispatch() in the event that the AEE_APP_SIG signal is set.

## Modifying the task loop

BREW uses AEE_APP_SIG to inform the BREW task that it needs to run. Therefore, the task loop must be modified to handle the BREW signal and invoke AEE_Dispatch whenever AEE_APP_SIG is set.

## Initializing BRIDLE SWI Handler

The function bridle_InitSWITable must be called before BREW is initialized, and regardless of whether BRIDLE is enabled. This step is necessary to populate the function pointer tables needed for uniform access between BRIDLE and non-BRIDLE builds. For information about BRIDLE, see Memory Security Through BRIDLE on page 94.

## BREW initialization

To initialize BREW, call AEE_Init() with a unique signal value for the BREW task. This function returns the pointer to IShell for you to save. It is used for calls to the IShell interface. You can call AEE_GetShell() to obtain the same pointer. You must invoke AEE_Init() in the context of the BREW task.

Within the AEE_Init() function call, BREW performs a number of tasks:

- Initializes IShell, IOEMDisplay, and ITAPI, among other interfaces.
- Calls OEM_GetConfig() with the CFGI_AUTOSTART selector to determine whether an applet needs to launch automatically after BREW completes initializing. The returned value is the ClassID of the applet that needs to auto-launch, or 0 (zero) if no applet is selected to run after the initialization. Typically, this value must be set to 0 (zero). However, there are cases where you can select to auto-launch an applet at startup. For example, if the device’s UI is written as a BREW applet, this applet is launched at the initialization.
- BREW sets a user-level timer by way of clock services if any alarms have been registered through ISHELL_SetAlarm(). This allows BREW to dispatch alarms and SMS messages to applications that have registered to receive these events. These functions are supported until the BREW AEE_Exit() function has been made. For this reason, the recommended procedure is for BREW to initialize when the device starts, and terminate when the device is about to power off. Although you can integrate the BREW environment so that it starts and ends on an as-needed basis, doing so prevents the proper operation of the SMS and clock services. Applications may not receive BREW alarms, which impacts applications such as calendar applications. More importantly, the BREW SMS functions do not function correctly. This prevents BREW applications from receiving SMS messages when BREW is not running. This also prevents the carrier from asynchronously recalling undesired applications from the devices.

**NOTE:** For DMSS releases that already include BREW, AEE_Init() is called from ui_init in ui.c.

### Troubleshooting

If this function returns NULL, take the necessary steps to recover because BREW has failed to initialize. The following can cause BREW not to initialize:

- OEM display initialization (OEMDisp_New) fails, returns an error code, and fills the OEM display interface pointer with NULL.
- ITAPI fails to initialize.
- The file system is corrupt or not working correctly.

## Terminating BREW

To terminate BREW, call AEE_Exit() before powering down the device. This is necessary because it allows BREW to free up resources and record a proper shutdown of BREW services. Failure to call AEE_Exit will cause the device to enter safe mode after every two shutdowns, thereby severely impacting the user experience.

**NOTE:** For DMSS releases that already include BREW, AEE_Exit() is called from ui_handle_stop_sig() in uihsig.c.

## Suspending and resuming BREW

**NOTE:** In previous BREW versions, there was a recommendation to suspend and resume BREW by calling AEE_Suspend() and AEE_Resume(), respectively, whenever the native UI took control of the main display. This caused problems for the coexistence of BREW applications and non-BREW applications, however, which led to the alternative method described below.

It is recommended not to use of the functions AEE_Suspend() and AEE_Resume(). Instead, whenever you want to take control of the main display or keypad, a transient BREW application must be started. After this transient BREW application is started, drawing to the screen must be done in the context of the BREW application. This ensures that there is a BREW application running at all times and makes it seamless for the user to switch from BREW applications to native applications. This logic can be applied in all cases, such as handling incoming calls, handling incoming SMS messages, and the like. Refer to Integrating native UI applications within BREW devices on page 117 for details.

# Sending Events to BREW

This section contains information about sending key events and other events to BREW.

## Sending key events

It is extremely important to send key events to BREW at all times. When any event is sent to BREW, the return value indicates whether the event was processed by BREW.

In scenarios in which it is not possible to send key events to BREW, you must call the function AEE_Active whenever a key is pressed. This allows BREW to deal with the screen saver correctly.

For each key pressed and released, you must translate key input into an AEE Virtual Key (AVK) code and send an event via each of the following functions.

| | |
|---|---|
| AEE_KeyPress() (EVT_KEY_PRESS) | Send as soon as the key is pressed. |
| AEE_KeyRelease() (EVT_KEY_RELEASE) | Send as soon as the key is released. |
| AEE_Key() (EVT_KEY) | Send immediately after EVT_KEY_PRESS or immediately before EVT_KEY_RELEASE, depending on your preference. Some device models have key input responses when a key is pressed and others when released. This event is configured according to each OEM. |

**NOTE:** The three functions listed above take one input parameter, which is the translated AEE Virtual Key code of the AVKType enumeration type defined in AEEVCodes.h.

BREW sends key repeat events to BREW applications while a key is held. In addition, BREW sets appropriate timer(s) upon the receipt of EVT_KEY_PRESS, and sends EVT_KEY events until EVT_KEY_RELEASE is received from the application.

BREW starts an internal timer as soon as it receives EVT_KEY. This timer is used for sending repeated EVT_KEY events to the BREW application while the key is held. This timer is repeated until BREW receives the EVT_KEY_RELEASE from the OEM. For the same reason, it is extremely important the EVT_KEY_RELEASE is sent to BREW as soon as the key is released. For example, sending EVT_KEY to BREW but not sending EVT_KEY_RELEASE to BREW causes repeated EVT_KEY events to be sent to the BREW application.

Previous BREW versions supported an event EVT_KEY_HELD to indicate that a key was held. This event is now deprecated and no longer supported. Instead, repeated EVT_KEY events are sent to the BREW application to indicate that a key was held.

**NOTE:** For DMSS releases that already include BREW, key events are handed to BREW in uisbrew.c. When the device is in the BREW state, uistate_brew() processes all UI events, including key events. uistate_brew() calls ui_to_brew_event() to translate the device's key code into the BREW AVK type. This function then calls the AEE_KeyPress(), AEE_Key(), and AEE_KeyRelease macros.

## Examples

Following is an example scenario, an event flow from a BREW application perspective, and steps for handling key events from an OEM perspective.

### Scenario

1. Press key AVK_1.
2. Hold key AVK_1 for 1 minute.
3. Release key AVK_1.

### Event flow from a BREW application perspective

1. EVT_KEY_PRESS (wParam = AVK_1)
2. EVT_KEY (wParam=AVK_1, dwParam=0)
3. EVT_KEY (wParam=AVK_1, dwParam=KB_AUTOREPEAT)

   *The KB_AUTOREPEAT indicates to the application that this is a repeat of the same key.*
4. EVT_KEY (wParam=AVK_1, dwParam=KB_AUTOREPEAT)
5. EVT_KEY_RELEASE (wParam=AVK_1)

### Steps for handling key events from an OEM perspective

1. Call AEE_KeyPress (AVK_1) when AVK_1 is pressed.
2. Call AEE_Key (AVK_1) for EVT_KEY event.
3. Call AEE_KeyRelease (AVK_1) when AVK_1 is released.

   **NOTE:** The buttons that are used to increase and decrease volume, which are usually located on the side of a device handset, must be mapped to AVK_VOLUME_UP and AVK_VOLUME_DOWN, respectively.

## Configurable timers for key repeat events

The following files are used to configure timers for key repeat events.

| File | Description |
|---|---|
| AEEVCodes.h | Defines two new constants. |
| KB_AUTOREPEAT_START | Indicates the time delay before keypad EVT_KEY events begin auto-repeating. |
| KB_AUTOREPEAT_RATE | Indicates the time delay between EVT_KEY auto-repeat events. |
| OEMConfig.h | Defines the new structure and CFGI_KB_AUTOREPEAT configuration item. |
| OEMKBAutoRepeat | |
| • OEMKBAutoRepeat.dwStart | • Indicates the time in milliseconds between the time the first EVT_KEY event fires and the key repeats. If the value is 0, no auto-repeat behavior is supported.The default value, if a nonzero is returned, is KB_AUTOREPEAT_START. |
| • OEMKBAutoRepeat.dwRate | • Indicates the time in milliseconds between each repeated EVT_KEY. If the value is 0, a single EVT_KEY (repeat) fires. The default value, if a nonzero is returned, is KB_AUTOREPEAT_RATE. |

You may control this behavior by modifying the values of OEMKBAutoRepeat as follows:

```
case CFGI_KB_AUTOREPEAT:
{
    OEMKBAutoRepeat * par = (OEMKBAutoRepeat *)pBuff;
```

```
     if(!pBuff || nSize != sizeof(OEMKBAutoRepeat))
     return(EBADPARM);

     if(par)

     {

       par->dwStart = NNNN;
       par->dwRate = NNNN;
       return(0);

      }
}
```

## Application behavior

Keypad auto-repeat keys are transmitted to the application as follows.

EVT_KEY_PRESS

EVT_KEY (wParam = Key, dwParam = 0)

- BREW sets a timer to KB_AUTOREPEAT_START or the OEM-defined value, if it is nonzero.
- BREW does *not* auto-repeat if you indicate that auto-repeat is *off* (OEMKBAutoRepeat.dwStart = 0).
- When the timer expires, the following logic repeats while the key is held.

  EVT_KEY (wParam = key, dwParam = KB_AUTOREPEAT)

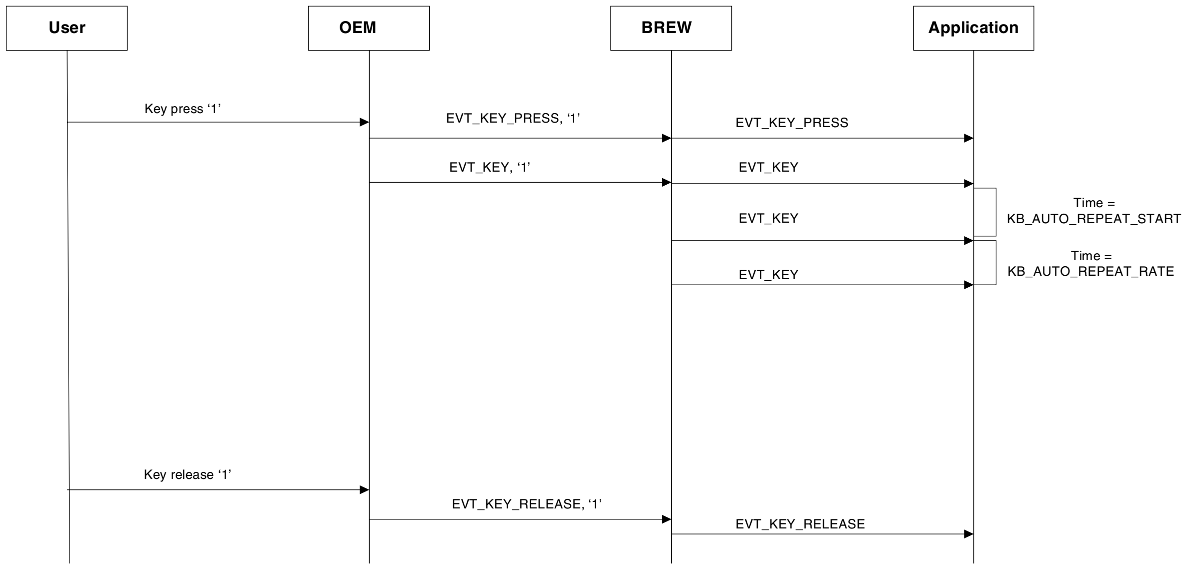
  The timer sets to KB_AUTOREPEAT_RATE or the OEM-defined value if it is nonzero.

  Events do not repeat if the OEM sets OEMKBAutoRepeat.dwRate to 0.

EVT_KEY_RELEASE

The following diagram shows a simple case in which a single key is pressed at one time.

### *Single keypress*



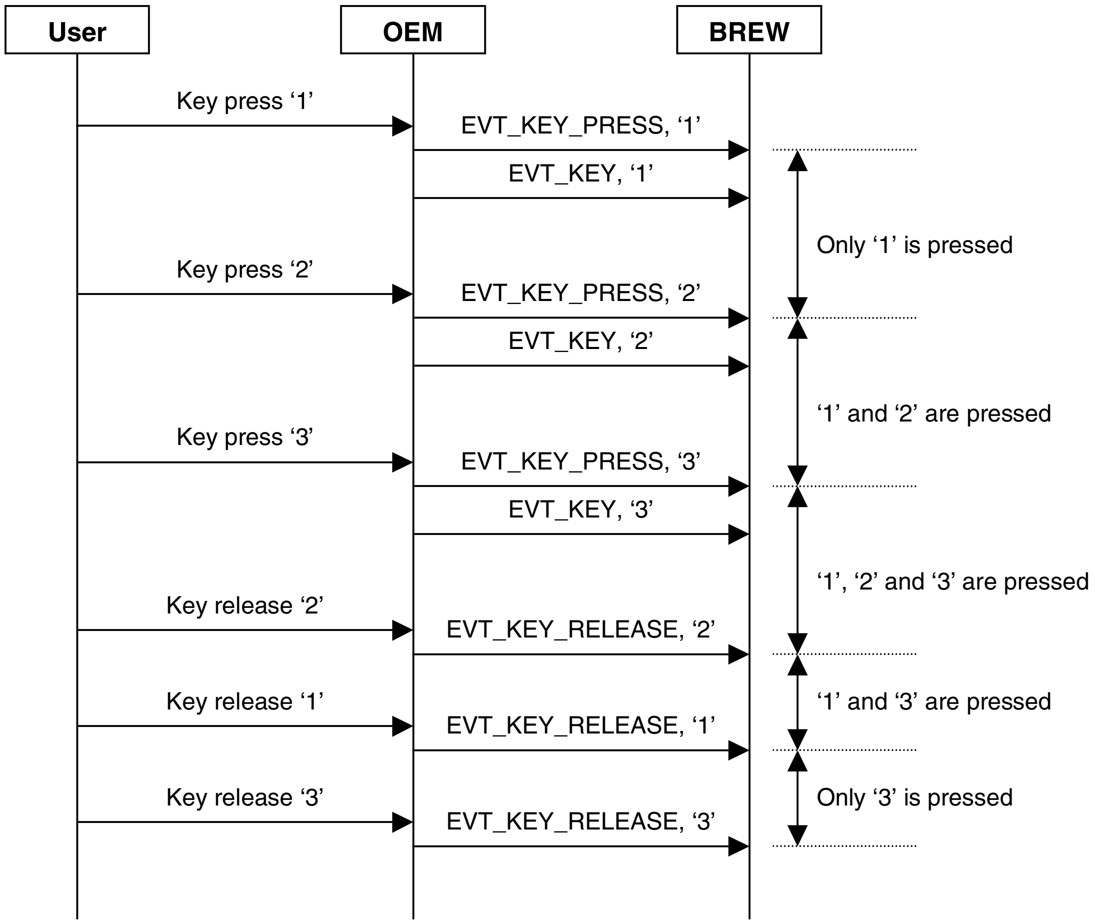
## Multiple keypress support

Multiple keypress support is used for advanced gaming and entertainment applications. When two or more keys are simultaneously pressed, no extra steps need to be taken. However, it is crucial that all the key events are reported to BREW in the correct order.

An important requirement is that the keypad has to detect multiple simultaneous keypresses and releases. Another requirement is to provide support for the device driver and hardware, and the OEM layer that relays the events to BREW correctly.

When two or more keys are pressed simultaneously, the same sequence of events holds true. The following diagram shows a case where three keys are pressed at one time.

### *Multiple keypress (three keys)*



Send all the events as key activities, as shown. These activities are detected by the device driver. It is the BREW applets that actually determine which keys are simultaneously pressed, based on the past EVT_KEY_PRESS and EVT_KEY_RELEASE events.

## Handling keyguard, flip, screen rotation, and headset

To handle keyguard, flip, screen rotation, and headset, you must send the following events to BREW:

- AEE_SEND_FLIP_EVT (p)
- AEE_SEND_KEYGUARD_EVT (p)
- AEE_SEND_HEADSET_EVT (p)
- AEE_SEND_SCRROTATE_EVT (p)

See the *BREW OEM API Reference* online help for documentation on these functions.

## Sending pen events

The addition of four events to AEE_Event allows you to send pen events through the OEM layer to BREW and BREW applications. Because these events specify positions or pen movement on the display, this capability offers support for devices such as PDAs that use a pen instead of a mouse. Three parameters are associated with these events: evt, wparam, and dwparam. The events and their parameters appear in the tables below.

### IAEE_Event pen events sent by OEMs

| | |
|---|---|
| AEE_Event() (EVT_PEN_DOWN) | Sent when the pen is put down on the display. |
| AEE_Event() (EVT_PEN_MOVE) | Sent when the pen is moved on the display. |
| AEE_Event() (EVT_PEN_UP) | Sent when the pen is lifted up from the display. |

### AEE_Event pen event sent by BREW

| | |
|---|---|
| AEE_Event() (EVT_PEN_STALE MOVE) | This pen event must not be sent by OEMs. BREW will send this event to the apps when it sees an EVT_PEN_MOVE event, and if there is a move near EVT_PEN_MOVE event in the queue. |

The four functions listed above each take three input parameters, evt, wparam, and dwparam.

**NOTE:** If there are between 30 and 39 pen events in the queue, half of the new pen events will be dropped. If there are between 40 and 44 pen events in the queue, two-thirds of the new pen events will dropped. If there are between 45 and 48 pen events in the queue, three-quarters of the new pen events will be dropped. If there are 49 pen events in the queue, four-fifths of the new pen events will be dropped. If there are 50 pen events in the queue, all of the new pen events will be dropped.

### Pen event parameters

| Parameter | Value | Description |
|---|---|---|
| AEEEvent evt | EVT_PEN_DOWN<br>EVT_PEN_MOVE<br>EVT_PEN_UP | Pen events |
| wparam (16 bit) | Ignored<br>**NOTE:** The wparam from the OEM layer is ignored and later filled in with the lower 16 bits of time. | Lower 16 bits of current time of day in milliseconds (ms) |
| dwparam (32 bit) | The pen position of the x coordinate on the display | Upper 16 bits = the signed x-coordinate |
| | The pen position of the y coordinate on the display | Lower 16 bits = the signed y-coordinate |

## Sending joystick events

The addition of two events to AEE_Event, EVT_JOYSTICK_POS (sent by OEMs) and EVT_JOYSTICK_STALE_POS (sent by BREW), allows you to send joystick events through the OEM layer to BREW and BREW applications. Three parameters are associated with these events: evt, wparam, and dwparam. The events and their parameters appear in the tables below.

### AEE_Event joystick events sent by OEMs

| | |
|---|---|
| AEE_Event()<br>(EVT_JOYSTICK_POS) | Sent when the joystick moves. |

### AEE_Event joystick event sent by BREW

| | |
|---|---|
| AEE_Event()<br>(EVT_JOYSTICK_STALE_POS<br>) | This joystick event must not be sent by OEMs. BREW will send this event to the apps when it sees an EVT_JOYSTICK_POS event, and if there is a move near EVT_JOYSTICK_POS event in the queue. |

**NOTE:** If there are between 30 and 39 joystick events in the queue, half of the new joystick events will be dropped. If there are between 40 and 44 joystick events in the queue, two-thirds of the new joystick events will dropped. If there are between 45 and 48 joystick events in the queue, three-quarters of the new joystick events will be dropped. If there are 49 joystick events in the queue, four-fifths of the new joystick events will be dropped. If there are 50 joystick events in the queue, all of the new joystick events will be dropped.

## *Joystick event parameters*

| Parameter | Value | Description |
|---|---|---|
| AEEEvent evt | EVT_JOYSTICK_POS | Sent when joystick moves |
| wparam (16 bit) | Ignored<br>**NOTE:** The wparam from the OEM layer is ignored and later filled in with the lower 16 bits of time. | Lower 16 bits of current time of day in milliseconds (ms) |
| dwparam (32 bit) | The joystick position of the x coordinate on the display<br>The joystick position of the y coordinate on the display | Upper 16 bits = the signed x-coordinate<br>Lower 16 bits = the signed y-coordinate |

# Implementing Display Support

Display management is the largest portion of porting BREW. You must implement this module first before moving on to other modules. For complete descriptions of all the functions needed for implementation, see th*e BREW™ OEM API Reference Online Help for MSM Platforms.*

## Core display information

The extended display layer provides access to multiple displays. A secondary display typically appears on the outside of a clamshell device. This section includes all new features of the display layer, especially the stock font support.

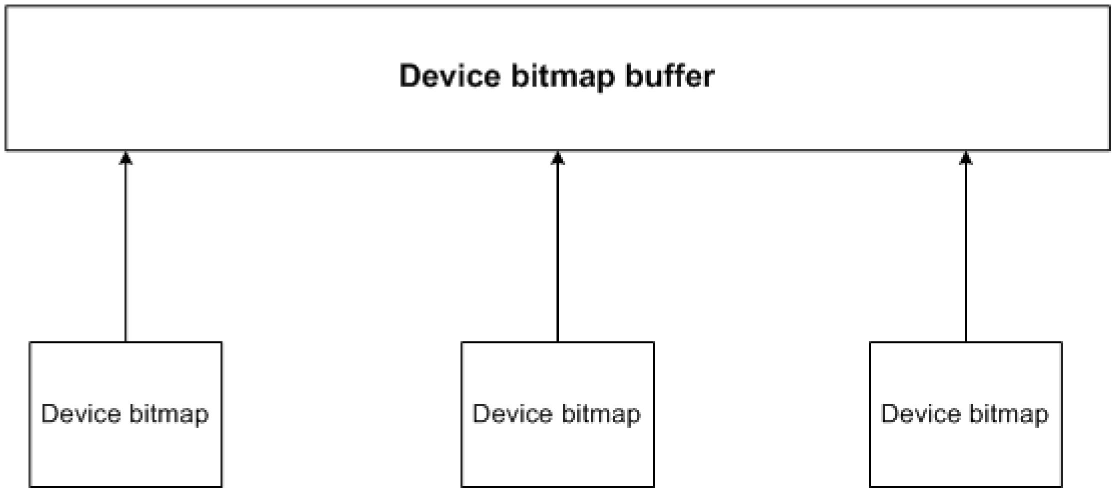
BREW accesses displays through the following three bitmap interfaces:

- IBitmap
- IBitmapDev
- IBitmapCtl

The IBitmap interface is exported by all bitmaps, and the IBitmapDev and IBitmapCtl interfaces are exported by device bitmaps.

Generally, you allocate static memory to hold the pixel data for each display. In BREW terminology, this is the device bitmap buffer; BREW expects that there is exactly one device bitmap buffer per display. A bitmap object exports an IBitmap interface and accesses a pixel buffer; if it accesses a device bitmap buffer, it is called a device bitmap. A display may have multiple device bitmaps, but all of the device bitmaps have a pointer to the same device bitmap buffer, as shown in the following diagram.

*Device bitmaps pointing to the same device bitmap buffer*



You should maintain one global device bitmap per display, obtained by BREW with AEECLSID_DEVBITMAPn.

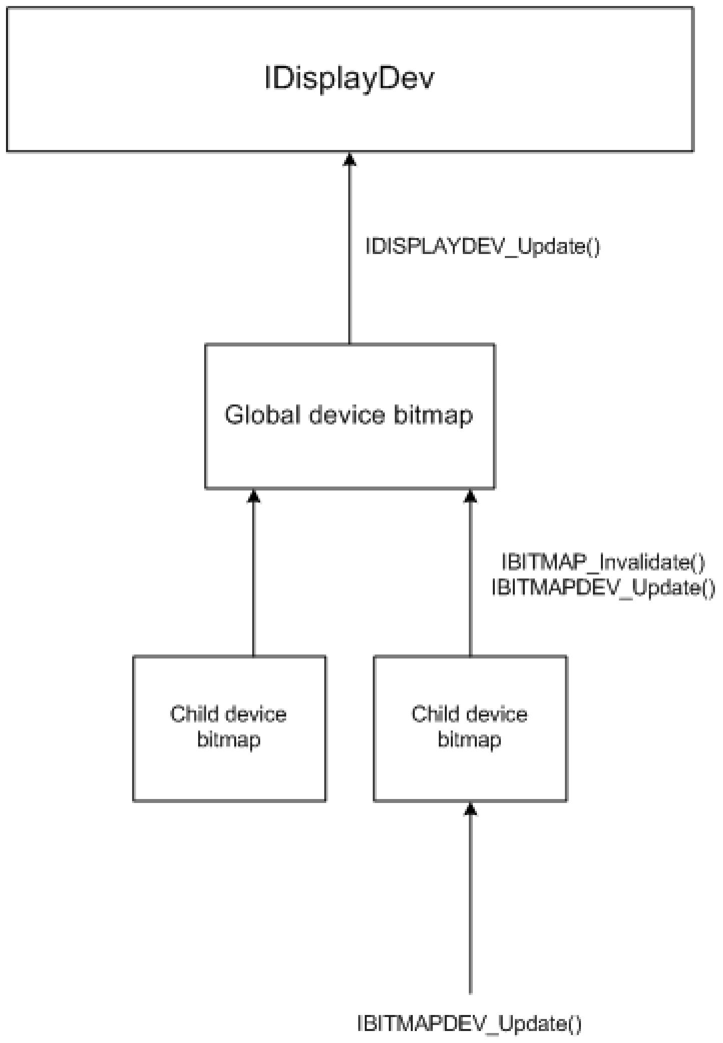
**NOTE:** Calling AEE_CreateInstanceSys() multiple times with AEECLSID_DEVBITMAP1 always returns the same IBitmap, AddRef'd.

AEECLSID_DEVBITMAPn_CHILD creates a child of the corresponding global device bitmap. You should always create a new device bitmap for each AEECLSID_DEVBITMAPn_CHILD request.

In BREW terminology, a display update refers to the process of copying all or part of a device bitmap buffer to the corresponding display, allowing you to see the image in the device bitmap buffer. A synchronous update is triggered by IBITMAPDEV_Update(), called by either BREW or a BREW application. In the case of a child device bitmap, IBITMAPDEV_Update() is implemented as a call to the global (parent) device bitmap's IBITMAPDEV_Update(). The reference implementation implements the global device bitmap's IBITMAPDEV_Update() by calling IDISPLAYDEV_Update(), where the work occurs.

The reference OEMBitmap implementation maintains a dirty rectangle for each device bitmap. This rectangle covers all of the pixels changed in the device bitmap since the last display update, allowing updates to be optimized by not copying the unchanged parts of the bitmap. Child device bitmaps pass their dirty rectangles to the global device bitmap by calling IBITMAP_Invalidate before calling IBITMAPDEV_Update(). This allows the global device bitmap's dirty rectangle to be expanded to include the child's dirty rectangle. The global device bitmap then passes this dirty rectangle as a parameter to IDISPLAYDEV_Update.

### Reference OEMBitmap implementation



In addition to the above-noted interfaces, you should put system fonts in the OEM mod table with the ClassIDs AEECLSID_FONTSYSNORMAL, AEECLSID_FONTSYSLARGE, and AEECLSID_FONTSYSBOLD. You should also implement AEE_DEVICEITEM_SYS_COLORS_DISPn and AEE_DEVICEITEM_DISPINFOn in OEM_GetDeviceInfoEx for each display.

Display access is granted differently for the primary display and any secondary displays. For the primary display, access is granted to the currently active BREW application. The OEM layer takes over the primary display by calling AEE_Suspend() and returns control of the display to BREW with AEE_Resume(). However, a recommended means to take control over the display is by starting another BREW application, called a shim application, and performing the drawing in the context of that application. See Integrating native UI applications within BREW devices on page 117 for details.

Secondary display access is granted first to the currently active application; if that application does not use the display, BREW moves down the active application stack and gives the display to the first application attempting to use it. If there is no application in the application stack trying to use the display, BREW moves through the background application list, starting with the most recent background application and ending with the application that has been running in the background the longest.

An application trying to use a display is defined as an application that has a reference to a device bitmap for that display. The OEM layer may take control of a secondary display by calling AEE_EnableDisplay() and return control of the display to BREW by calling AEE_EnableDisplay() again. For both the primary and secondary displays, BREW calls IBITMAPCTL_Enable() to enable or disable the application's instance of the device bitmap for a particular display. This triggers any callbacks registered with IBITMAPDEV_NotifyEnable().

## Display

IOEMDisp provides backlight and annunciator control. IOEMDisp contains methods for controlling other display functions that were used prior to BREW 3.0 but are now deprecated. Refer to OEMDisp.h for the interface specification and OEMDisp.c for a sample implementation.

IDisplayDev updates displays. Updating a display means copying the contents of a device bitmap to the corresponding display's frame buffer. You need one implementation of this interface for each display on the device that is accessible to BREW. Refer to OEMDisplayDev.h for the interface specification and OEMDisplayDev.c for a sample implementation.

## Bitmaps

You provide a bitmap implementation for each display type supported by the device. A bitmap object exports several interfaces. Every device compatible bitmap object must export the IBitmap and ITransform interfaces. These interfaces provide basic drawing functionality. Every device bitmap object must additionally export the IBitmapDev and IBitmapCtl interfaces. These interfaces provide functionality specific to device bitmaps. Sample bitmap implementations can be found in the following files:

- OEMBitmap_priv.h
- OEMBitmap_generic.h
- OEMTransform_generic.h

- OEMBitmap1.c
- OEMBitmap2.c
- OEMBitmap8.c
- OEMBitmap12.c
- OEMBitmap16.c
- OEMBitmap18.c

The .h files contain a generic implementation written in terms of macros defined in the .c files. Choose the .c file most similar to the required display format and modify that file only.

The IBitmap interface provides basic drawing functionality and is accessible by applications. It is defined in AEEBitmap.h.

The ITransform interface provides transformation blitting functionality and is accessible by applications. It is defined in AEETransform.h.

The IBitmapDev interface provides display-specific functionality and is accessible by applications. It is defined in AEEBitmap.h.

The IBitmapCtl interface provides display-specific functionality that is needed only by BREW and is not accessible to applications. It is defined in OEMDisp.h.

## Application frames

Applications specify certain display settings, either in their MIF files or during runtime, by passing these settings to IDisplay. These settings are in the form of a string passed by BREW to the IAppFrame interface. The settings currently defined by BREW are width, height, color depth, and whether the annunciator bar is displayed. There may be up to one application frame per display per application. Treat the display settings string as a request that does not always need to be granted. The one behavior is, that displays with color depths of greater than 16 bits, should default to a 16-bit DIB-compatible mode. The IAppFrame implementation grants a request for maximum display depth, which allows applications to use the native color depth of the display. In this case, IAppFrame switches the mode of the device bitmap for the display. A function is provided for this in OEMBitmap called OEMBitmap_CopyMode(). Your IDisplayDev implementation must support updating from device bitmaps in each desired mode.

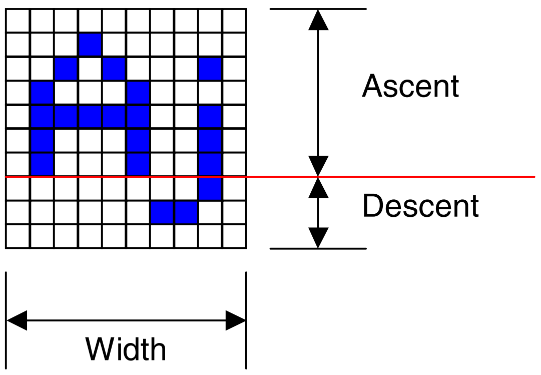
The IAppFrame interface is defined in OEMAppFrame.h. A sample implementation is provided in OEMAppFrame.c.

## Fonts

IFont provides the text-related functions. The following functions measure the dimension of the text or the font:

- IFONT_GetInfo(): This function returns the vertical measurement of the font. This measurement is divided into two sections: ascent and descent. The ascent is the number of pixels that can extend above the baseline, whereas the descent is the number of pixels that can extend below the baseline. For example, the example font shown below, A j, has the ascent and descent values of 7 and 3, respectively.

### *IFONT_GetInfo() interface example*



- IFONT_MeasureText(): This function measures the width of a string of text in pixels. For the IFONT_GetInfo() interface example (A j), the result of this function is 10.

Usually, IFONT_DrawText() is preceded by one or both of the above functions. IFONT_DrawText() takes in the input string in the wide character, 2-byte per character format. This is required for some foreign language encoding types (see Encoding type support on page 42).

## BREW Bitmapped Fonts

An IFont implementation of BBF is provided in aeefont.lib. To use this implementation you must generate a BBF file with the BBFGEN tool as described in *BREW PK Utilities Guide*. After obtaining the BBF data, you must load it into your build and the AEEBitFont_NewFromBBF() API.

There are also several sample BBF fonts in aeefont.lib that can be enabled on your device by defining the feature FEATURE_BREW_FONTS.

## Encoding type support

Although BREW's internal string encoding type is Unicode (2 bytes per character), BREW supports an array of encoding types including the following through utility functions:

- EUC-CN (GB2312, simplified Chinese)
- EUC-KR (CP949 and KSC5601, Korean)
- Shift-JIS (JIS, Japanese)

To accommodate BREW's internal encoding type, each string going into BREW needs to be 2 bytes per character. For this reason, the function STREXPAND() is provided to expand strings of the above encoding types into hybrid strings that consist of all wide characters (that is, 2-byte characters). The encoding values don't change; only the total number of bytes occupied by each character is adjusted to a constant 2 bytes. To accomplish this, BREW inserts 0x00 in front of all single byte characters and leaves double-byte characters alone, as shown in the following string:

**NOTE:** This example assumes the target is based on Little Endian. The vertical lines ("l") are used as arbitrary character separators.

World のビジ
Shift-JIS: 0x57 l 0x6f l 0x72 l 0x6c l 0x64 l 0x82 0xcc l 0x83 0x72 l 0x83 0x64
BREW Hybrid: 0x5700 l 0x6f00 l 0x 7200 l 0x6c00 l 0x6400 l 0xcc82 l 0x7283 l 0x6483

Depending on the encoding type returned by wEncoding of the OEM_GetDeviceInfo() function call, STREXPAND() is adjusted as the following table shows.

| Language | BREW encoding type | Character set |
|---|---|---|
| Simplified Chinese | AEE_ENC_EUC_CN | GB2312 |
| Korean | AEE_ENC_KSC5601 | KSC5601 |
|  | AEE_ENC_EUC_KR | CP949 |
| Japanese | AEE_ENC_S_JIS | JIS |

If you are creating your own string, you need to explicitly call out STREXPAND() to expand the string. If you are getting a string of characters from a BREW Applet Resource (BAR) file, this step is already taken and you get a hybrid text string.

After the strings reach the display functions OEMDisp_DrawText() and OEMDisp_MeasureText(), contract them to the original form. This task is performed by the WSTRCOMPRESS() function. Then, depending on the device's encoding type, this function follows the rules for one of the following: KSC5601 (AEE_ENC_KSC5601), CP949 (AEE_ENC_EUC_KR), S-JIS (AEE_ENC_S_JIS), or GB2312 (AEE_ENL_FUC.CH). After this function is performed, the returned string is in the original KSC5601 or CP949 encoding type for Korean, Shift-JIS encoding type for Japanese, or GB2312 encoding type for Chinese.

## PNG support

In previous releases, the PNG library (libpng) and the BREW PNG support were integrated into a single library (AEEPNG.lib). If you have your own libpng, support is now provided by splitting AEEPNG.lib into the following libraries (see Scenarios on page 43 for guidelines to follow):

- AEEPNG.lib: Contains only the BREW-specific sources.
- PNG.lib: Contains the standard LIB PNG sources. This corresponds to the libpng library version 1.2.5.

### Scenarios

If you are not using libpng in a non-BREW environment to support PNG in BREW you need to link with two libraries that come with the Porting Kit (AEEPNG.lib and PNG.lib).

The following procedure applies if you already use libpng in a non-BREW environment.

### To support PNG in a non-BREW environment

1. Be sure, when your PNG.lib is built, that the following compiler-defines are defined inside the file pngconf.h:

```
#define PNG_USER_MEM_SUPPORTED
#define PNG_FLOATING_POINT_SUPPORTED
#define PNG_SETJMP_NOT_SUPPORTED
#define PNG_NO_ZALLOC_ZERO
#define PNG_NO_READ_cHRM
#define PNG_NO_READ_hIST
#define PNG_NO_READ_sPLT
#define PNG_NO_READ_tIME
#define PNG_NO_READ_UNKNOWN_CHUNKS
#define PNG_NO_READ_USER_CHUNKS
#define PNG_NO_READ_iCCP
#define PNG_NO_READ_iTXt
#define PNG_NO_READ_sCAL
#define PNG_NO_MNG_FEATURES
#define PNG_NO_FIXED_POINT_SUPPORTED
```

2. Rebuild the png.lib.

3. Link the device image with png.lib produced after completing steps 1 and 2, above, with the AEEPNG.lib that comes with the Porting Kit.

   **NOTE:** You can ignore the PNG.lib that comes with the Porting Kit.

## Annunciators

The annunciator bar on the phone can be defined as a separate area of the display that provides information about such things as battery status, RSSI, network status, and new messages. One of the ways to implement annunciators is using the BREW widgets and forms. More information about widgets and forms can be found in the documentation shipped with the widgets and forms package. The annunciator bar can also be defined as part of the Rootform (a widget and forms term) or can be part of a particular applet (depending on the UI requirements/design). OEMs can also choose to have an annunciator bar area separate from the application display area.

The various annunciators on the phone can be defined as individual widgets that listen to the appropriate models for change in status. For instance, you can define a MessageAnnunModel that has state information about new voicemail and pages SMS, EMS and MMS messages. It can also have state information about message send (i.e. sending such messages as SMS, EMS, MMS). The messaging widget (or widgets) can register a listener with this model so that they can be notified about state changes. The widgets can then choose to update themselves appropriately on a state change notification from the model.

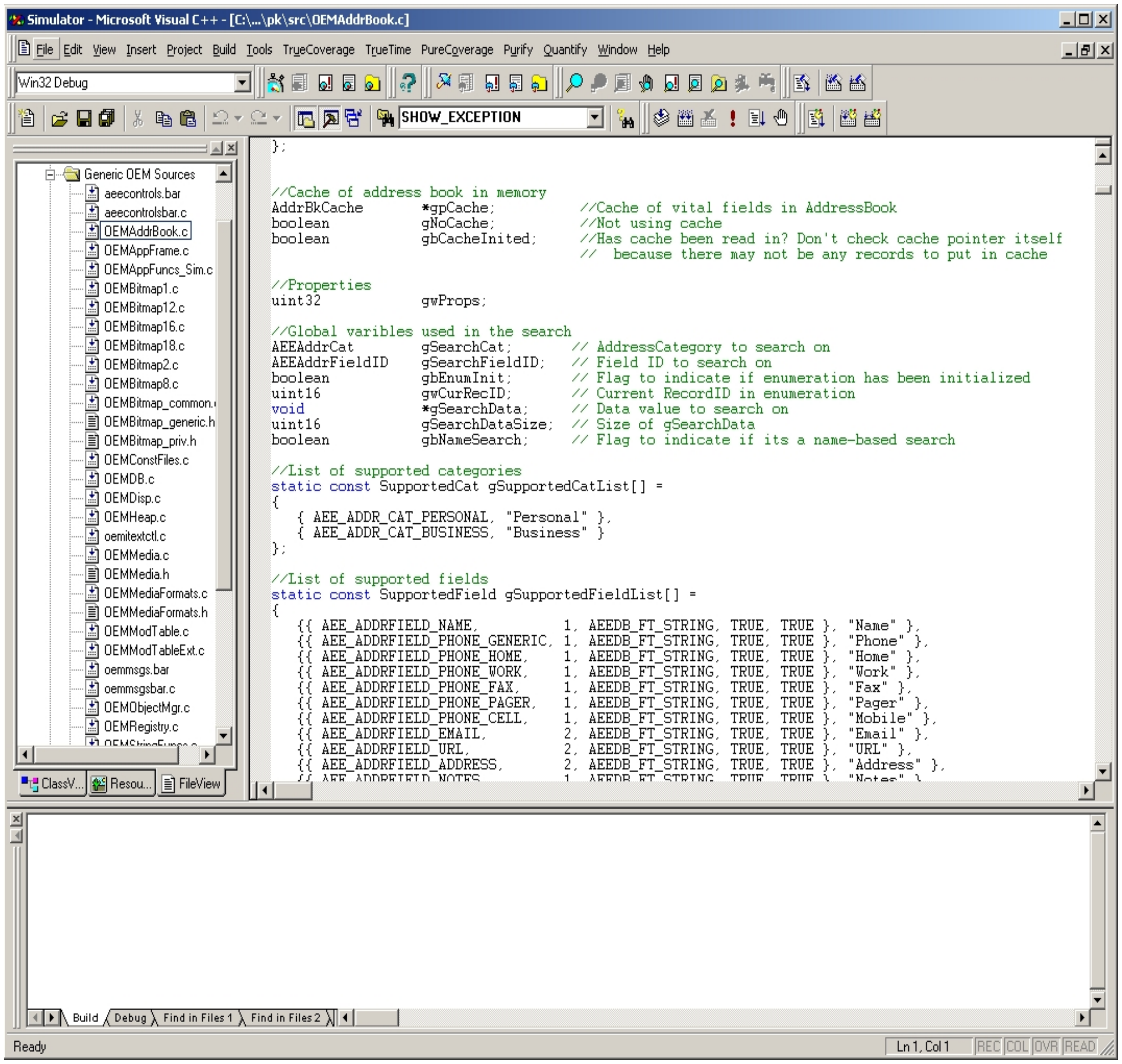
See AEESMSMsgModel.c and AEESMSMsgModel.h. These files implement ISMSMsgModel which maintains message status for SMS/EMS/MMS send/receive, used by the messaging annunciator widget.

## Using the Brew Simulator

The Porting Kit includes the BREW Simulator for Windows environments, and also the Microsoft Visual Studio workspace for Simulator, which includes:

- All the OEM sources that can be modified
- References to several libraries, including BREWWin.lib (BREW library for Windows), with which the Simulator links before building the executable

An example follows.



```
};


//Cache of address book in memory
AddrBkCache       *gpCache;          //Cache of vital fields in AddressBook
boolean           gNoCache;          //Not using cache
boolean           gbCacheInited;     //Has cache been read in? Don't check cache pointer itself
                                     //  because there may not be any records to put in cache

//Properties
uint32            gwProps;

//Global varibles used in the search
AEEAddrCat        gSearchCat;       // AddressCategory to search on
AEEAddrFieldID    gSearchFieldID;   // Field ID to search on
boolean           gbEnumInit;       // Flag to indicate if enumeration has been initialized
uint16            gwCurRecID;       // Current RecordID in enumeration
void              *gSearchData;     // Data value to search on
uint16            gSearchDataSize;  // Size of gSearchData
boolean           gbNameSearch;     // Flag to indicate if its a name-based search

//List of supported categories
static const SupportedCat gSupportedCatList[] =
{
   { AEE_ADDR_CAT_PERSONAL, "Personal" },
   { AEE_ADDR_CAT_BUSINESS, "Business" }
};

//List of supported fields
static const SupportedField gSupportedFieldList[] =
{
   {{ AEE_ADDRFIELD_NAME,          1, AEEDB_FT_STRING, TRUE, TRUE }, "Name" },
   {{ AEE_ADDRFIELD_PHONE_GENERIC, 1, AEEDB_FT_STRING, TRUE, TRUE }, "Phone" },
   {{ AEE_ADDRFIELD_PHONE_HOME,    1, AEEDB_FT_STRING, TRUE, TRUE }, "Home" },
   {{ AEE_ADDRFIELD_PHONE_WORK,    1, AEEDB_FT_STRING, TRUE, TRUE }, "Work" },
   {{ AEE_ADDRFIELD_PHONE_FAX,     1, AEEDB_FT_STRING, TRUE, TRUE }, "Fax" },
   {{ AEE_ADDRFIELD_PHONE_PAGER,   1, AEEDB_FT_STRING, TRUE, TRUE }, "Pager" },
   {{ AEE_ADDRFIELD_PHONE_CELL,    1, AEEDB_FT_STRING, TRUE, TRUE }, "Mobile" },
   {{ AEE_ADDRFIELD_EMAIL,         2, AEEDB_FT_STRING, TRUE, TRUE }, "Email" },
   {{ AEE_ADDRFIELD_URL,           2, AEEDB_FT_STRING, TRUE, TRUE }, "URL" },
   {{ AEE_ADDRFIELD_ADDRESS,       2, AEEDB_FT_STRING, TRUE, TRUE }, "Address" },
   {{ AEE_ADDRFIELD_NOTES,         1, AEEDB_FT_STRING, TRUE, TRUE }, "Notes" }
```

## Verifying implementation

Use OAT to verify your implementation of the display interfaces. See the *BREW™ Porting Evaluation Kit Test Case Online Help* in the PEK documentation set for details.

# Understanding the Generic Serial Interface

Serial I/O (SIO) in a PC involves communicating with an external device by connecting it to the PC's 9-pin connector. This enables an external device to communicate with the PC software by using serial communication. The same concept, applied to BREW, allows a variety of devices to communicate with BREW. The SIO closely models the Windows SIO model, while introducing a plug-and-play mechanism for BREW to detect any connected BREW devices.

In general, mobile manufacturers configure mobile devices in data mode, in which an AT command processor (ATCOP) accepts standard AT-style modem commands. The mobile device acts like a modem to the laptop and initiates data service connections in response to ATDxxx dialing commands. When a data connection is established, data to and from the laptop is passed through unmodified, and the ATCOP is then out of the loop. In this way, a laptop connects by using the phone as a modem.

By enabling BREW SIO, the device communicates with a BREW entity, such as a dynamic application or a BREW internal object. Once the link is established between the application and the device, it determines the protocol with which to facilitate communication. The BREW SIO acts as a dumb pipe.

Two aspects of the BREW SIO are based on the initiating party. The management of the connection setup varies based on the initiator. The device-initiated service involves a well-defined protocol to discover which application or internal entity services the device.

## Device-initiated service

When a device is connected, it will initially communicate with the ATCOP. By issuing a command, the device informs the ATCOP to transfer the control of the particular SIO connection to the BREW SIO Command Processor (BSCOP). When the device gets a positive response from BREW, it issues commands to the BSCOP. These commands allow the device to communicate with a BREW application or perform other tasks.

## Application-initiated service

BREW SIO also allows an application to unilaterally seize control of a serial port. This action succeeds or fails depending on what other client is currently active on that port. ATCOP and BSCOP usually yield to a requesting application, but another client, such as service programming, might refuse to release the port. The situations that prevent an application from gaining control of the port differ from OEM to OEM.

Application-initiated connections may be necessary to initiate communication with devices that are not BREW-aware. In application-initiated scenarios, however, the user must somehow coordinate connecting the device with launching the appropriate application.

## Application design considerations

Some application design considerations are discussed in the following paragraphs.

### Disconnection of a device while talking to an application

In device-initiated service, if the device is disconnected, the port reverts to the ATCOP. Any further read/write calls to IPort by the application result in errors. The application can reregister for a new connection or a reconnection of the previous port by calling Writeable().

In application-initiated service, if the device is disconnected, the application owns the port. Any further read/write calls result in errors. However, the application could give control of the port back to the ATCOP or retain it to do more work.

### Exiting an application during device communication

The application closes the serial port object; this action causes the port to revert to the ATCOP. If the application is reentered, the usual process of obtaining a serial port takes place.

### General application behavior with unexpected data

Applications that explicitly open a serial port must be mindful of the normal functionality of BSCOP and ATCOP and respond appropriately when connected to devices that expect to communicate with ATCOP or BSCOP. In general, the application should close the port and let BREW decide the next action.

DTR transition is the method used by the UARTs to detect device disconnections. In some cases, reliable detection may not be possible; for example, when an application is communicating to a particular device and another device replaces it. It is a good practice on the part of the application to detect this change due to errors it encounters and close the port so the control reverts to ATCOP.

## Using the BSCOP

The following table includes descriptions of commands issued by the connected device when in BSCOP mode. These commands allow the BSCOP to initiate communication with a BREW application.

| Command | Description | Responses |
|---|---|---|
| $BREW | AT$BREW is the command to the mobile device's ATCOP to transfer control to BREW. If BSCOP is already in control, this is interpreted as a $BREW command with a tag of AT, and the resulting response packet, including the tag, is ATOK.<br><br>As a result, when AT$BREW is sent to initiate communication, the device synchronizes, whether the port was in ATCOP or BSCOP mode. | OK—The mobile device ATCOP's response to hand the SIO to BREW.<br><br>ERROR: The mobile does not understand the AT$BREW command (for example, No BREW SIO at that particular port). |

| Command | Description | Responses |
|---|---|---|
| DEV:<devid>:<args> | This command initiates communication with a BREW application or object. BREW tries to find the handler using the identifier string. If BREW finds the handler, the START response is issued to the device. On failure, the ERROR response is issued.<br><br>The <devid> value is the registry key used to find the application handler. These keys should be of a regular form, such as <company code>-<devicename>, to avoid naming conflicts. The devid is limited to the printable ASCII characters excluding "*" (colon).<br><br>The <args> value will be passed to the launched application. <args> value is a string of bytes excluding the <CR> and <LF> characters. | OK—The command to the device indicating that the handler is found, and the application is launched. When the START command is issued, the device and the BREW entity are connected and ready to communicate using their predefined protocol.<br><br>ERROR:<xxxx>—Could not launch handler. <xxxx> gives the error code (from AEEError.h) as four hexadecimal digits. Possible values specific to SIO include: AEE_SIO_NOHANDLER (handler was not found). Other values, such as ENOMEMORY, are always possible. |
| VER | This command gets the BREW version. | OK:<ver>—<ver> = BREW version string, in a x.y.z.b. format (for example, 1.0.1.18). |
| APP:<clsid>:<args> | This command gives the CLSID of the application to open. BSCOP proceeds to launch an application as it does with the DEV command, although its ClassID, instead of a handler lookup, specifies the application. This is less extensible than the DEV command, but it is useful for debugging and development. The <clsid> is a string of hexadecimal letters that are constructed into a BREW ClassID. <args> is the same as defined in DEV. | OK—As in DEV.<br><br>ERROR:<xxxx>—As in DEV. |

| Command | Description | Responses |
|---|---|---|
| URL:<url> | BSCOP calls ISHELL_Browse URL() with the named URL. This launches a browser, MobileShop™, or some other application, depending upon which application has registered support for the URL scheme. After failing to launch a required application, a device could use MobileShop URLs to point the user to the required download option. <url> is of same format as the DEV: <args>. | OK—Indicates that the associated application was launched.<br>ERROR:<xxxx>—Indicates that an associated application could not be launched. Any error in AEEError.h was not returned, but, in particular, the following are most likely:<br>• ESCHEMENOTSUPPORTED (a BREW error code)<br>• ENOMEMORY |
| END | Informs BREW to relinquish command to the mobile ATCOP. | OK—This is the only expected response for the command. |

## Command and response framing

Each command is contained in a packet that begins with a 2-byte tag and ends with a <CR> (ASCII 0x0D) character. An <LF> (ASCII 0x0A) character following a command packet is ignored. Response packets begin with a 2-byte tag and end in <CR><LF> (ASCII 0x0D 0x0A). The maximum packet size supported by BSCOP is 512 bytes.

Tags sent with commands should consist of two alphanumeric ASCII characters. The tag attached to a response is the same as the tag sent with its corresponding command. Devices use this mechanism to disambiguate responses. By sending a different tag with every command, the device determines from which command a response results. This is useful in synchronizing communication when establishing the connection or recovering from data errors.

## Examples of BSCOP command sequences

Lines starting with D: represent data sent by the device, and P: represents data sent by the handset to the device.

D: 01AT$BREW

P: 01ATOK

D: 02VER

P: 02OK:3.0.0.1

D: 03DEV:BREW.siotest

P: 03ERROR 0C01

D: 04DEV:BREW.siotest

P: 04OK

D: 99END

P: 99OK

The device is not required to wait for a response before sending another command.  For example, this sequence could occur:

D: 02VER

D: 03DEV:kb

P: 02OK:3.0.0.1

P: 03OK

## IPort interface

A new BREW interface models duplex communications. This interface extends the ISource interface by adding the Write and Writeable members as shown:

```
AEEINTERFACE(IPort){
   INHERIT_ISource(IPort);
   Int    (*GetLastError)(IPort * po);
   int32  (*Write)(IPort *pme, char *pBuf, int32 cbBuf);
   void   (*Writeable)(IPort *pme, AEECallback *pcb);
   int    (*IOCtl)(IPort *po, int nOption, uint32 dwVal);
   int    (*Close)(IPort * po);
   int    (*Open)(IPort * po, const char * szPort);
};
```

The GetLastError() function reports the last error that occurred during the operation of the IPort. The return value is one of the global BREW error codes defined in AEEError.h. The Open() function allows the application to bind the IPort to a physical port.

When an instance of AEECLSID_SERIAL is created, an IPort is returned that is not associated with any physical port. IPORT_Open() indicates the name of the desired port.

Open() is a non-blocking call that might return AEEPORT_WAIT when it cannot be immediately satisfied. The caller then uses IPORT_Writeable() to receive a notification of subsequent attempts.

When calling Open(), the caller indicates the serial port desired by a zero-terminated string containing its name. BREW defines some names for types of ports that are generally available across many devices. Serial port names consist of short ASCII sequences, allowing different mobile devices to support different ports in an extensible manner. Usually the main port at the bottom of a phone is an UART. All the UARTs are represented with strings, AEE_PORT_SIO1(PORT1), AEE_PORT_SIO2(PORT2), and the like. The USB ports are represented using USB1, USB2, and the like. BREW also defines a special name, AEE_PORT_INCOMING (inc), that establishes a link with a device attempting communications with an application.

## Device-initiated usage

If a device is connected to BREW, and BREW starts an application based on the DEV: stringt, the application needs an IPort to communicate with the device.

When an application capable of communicating using SIO starts, it creates an IPort interface using the CLSID of AEECLSID_SERIAL, and then calls Open() with AEE_PORT_INCOMING. If Open() returns AEEPORT_WAIT, the application waits for the device-initiated connection by registering a callback using Writeable(). When a device is connected, the Writeable callback is called, prompting the application to retry the Open() operation, which succeeds.

If you start the application without connecting the device, the application follows a similar process and waits until the device is connected. This way, a device connected after its application launches is still connected. Until the device is connected, Open() continues to return AEEPORT_WAIT, and Writeable() does not fire.

AEESIO_PORT_INCOMING applies only to devices that request the running application. If one application requests AEESIO_PORT_INCOMING and a device is then connected that requests a *different* application, the first application's Open() is not satisfied. Instead, the other application launches, and its attempt to open AEESIO_PORT_INCOMING succeeds.

AEESIO_PORT_INCOMING refers to any serial port. Imagine a phone with multiple UARTs or multiple USB virtual serial ports, each of which accepts device-initiated connections.

## Application-initiated usage

The application creates an IPort interface using the Open() function. The port string argument determines which port opens. The port IDs supported by BREW are given in AEESio.h. For example, to open the main serial port, the AEESIO_PORT_SIO1 string is used.

The Open() could fail due to multiple reasons such as nonavailability (service programming in progress, Mobile busy, no-permission for open), no such port, and the like. In this case, the Writeable callback is called, and a call to GetLastError() reports the error particulars.

## Closing a port

When an IPort is closed, it is dissociated from the physical port, and the port is returned to the ATCOP.

Port objects are closed implicitly when all references to the object are released, but the Close() function allows explicit closing. This function is convenient when different layers or modules in the system use the same port object. This function also allows an IPort to be reused, because when it is in the closed state, Open() can be called again.

## Serial port configuration

The IOCtl flags, AEESIO_IOCTL_SCONFIG and AEESIO_IOCTL_GCONFIG, set and get configuration using the AEESIOConfig data structure as defined in AEESio.h. AEESIOConfig has information to control a UART such as baud rate, parity, stop bits, and the like. In the case of virtual serial ports, such as USB-based virtual serial ports, some or all of these settings might be ignored. As all entries of the AEESIOConfig may not be supported by an implementation the IPort, the return value of SUCCESS may not mean that all options are set. Getting the configuration, after setting it, returns the current changed configuration. For example, if a specific baud rate cannot be set, the nearest supported baud rate is set. Setting a baud rate to 38500 may actually set the real configuration to the nearest supported baud rate of 38400.

The IOCtl also supports options to adjust internal buffer sizes, setting triggers (minimum number of bytes before making the state readable, and the like) or doing efficient reads.

### Application registration for supported devices

Applications that handle specific devices must register with BREW so they can be informed on request. This registration information is stored in the app's MIF, which can be updated using the MIF Editor.

#### To update application registration information

1. In the MIF Editor, click **Extensions** and **New** in the Exported MIME types section.
2. Enter the device id string in the MIME Type held.
3. Enter the handler type in the base class.

   **NOTE:** The handler type for SIO devices is defined in the AEESio.h as AEECLSID_HTYPE_SERIALDEVICE (0x01011be6). The handler ClassID is the same as the app's CLSID.

## Using DMSS changes to enable BREW SIO

QUALCOMM's DMSS chipset software supports SIO, with a limitation that it is not readily exposed to application-level software such as BREW. This document lists the required DMSS changes based upon 6050 chipset DMSS software. As your target chipset may be different, the following instructions are only a case study.

There are two primary tasks for enabling SIO:

- Enable the Runtime Device Mapper (RDEVMAP) system to allow dynamic services and BREW services to control the ports.
- Enable an AT command AT$BREW in the ATCOP that, when called, transfers the control of the port to the BREW service.

The changes are in two subsystems, the RDEVMAP and the ATCOP.

The RDEVMAP is a service implemented in rdevmap.c that manages the distribution of devices (Serial, USB ports) to requested parties by providing an API. BREW SIO uses this feature to acquire and release ports.

**NOTE:** This feature must be enabled in DMSS by defining FEATURE_RUNTIME_DEVMAP.

BREW SIO implementation also requires changes in the ATCOP. When the AT$BREW command is issued to the ATCOP, it gives control of the port to BREW. The changes in the ds* files listed below facilitate this behavior.

RDEVMAP changes are in the following files:

- rdevmap.h
- rdevmap.c

ATCOP changes are in the following files:

- dsatcop.h
- dsatcopi.h
- dsatcop.c
- dsatdat.c

**NOTE:** Later versions of DMSS made the ATCOP changes easier.

## File changes

File changes are shown in the context of existing code. The Before and After sections show complex changes. In some cases, changes are highlighted as part of the existing code.

The following examples are based on the MSM6050 header and source files. Line numbers may vary.

### File: rdevmap.h

**Add a new BREW serial port service to rdm_service_enum_type**

Line: 99

*Before:*

```
/*-------------------------------------------------------------------------
  Type that defines the Services that can utilize a serial port
-------------------------------------------------------------------------*/
typedef enum
{
  RDM_NULL_SRVC = -1,         /* The NULL (no) service               */
  RDM_DIAG_SRVC = 0,          /* DIAG Task                             */
  RDM_DATA_SRVC,              /* Data Service Task                     */
  RDM_BT_HCI_SRVC,            /* Bluetooth Task                        */
  RDM_MMC_SRVC,               /* MMC over USB                          */
  RDM_NMEA_SRVC,              /* NMEA service                          */
#ifdef FEATURE_ONCRPC
  RDM_RPC_SRVC,               /* ONCRPC task                           */
#endif
  RDM_SRVC_MAX                /* Last value indicator (must be last) */
} rdm_service_enum_type;
```

*After:*

```
/*---------------------------------------------------------------------------
  Type that defines the Services that can utilize a serial port
---------------------------------------------------------------------------*/
typedef enum
{
  RDM_NULL_SRVC = -1,         /* The NULL (no) service                 */
  RDM_DIAG_SRVC = 0,          /* DIAG Task                             */
  RDM_DATA_SRVC,              /* Data Service Task                     */
  RDM_BT_HCI_SRVC,            /* Bluetooth Task                        */
  RDM_MMC_SRVC,               /* MMC over USB                          */
  RDM_NMEA_SRVC,              /* NMEA service                          */
#ifdef FEATURE_ONCRPC
  RDM_RPC_SRVC,               /* ONCRPC task                           */
#endif

  RDM_DYNAMIC_SRVC1,         /* Dynamic service                      */
  RDM_DYNAMIC_SRVC2,         /* Dynamic service                      */
  RDM_DYNAMIC_SRVC3,         /* Dynamic service                      */
  RDM_DYNAMIC_SRVC4,         /* Dynamic service                      */
  RDM_DYNAMIC_SRVC5,         /* Dynamic service                      */
  RDM_DYNAMIC_SRVC_LAST,     /* Dynamic service                      */

  RDM_SRVC_MAX                /* Last value indicator (must be last) */
} rdm_service_enum_type;
```

### New function prototypes

Add the following at the end of the rdevmap.h file.

```
sio_port_id_type rdm_register_new_service( rdm_device_enum_type dev,
                    rdm_service_open_func_ptr_type  open_fn,
                    rdm_service_close_func_ptr_type close_fn,
                    rdm_service_enum_type          *pService);

void rdm_unregister_service(rdm_service_enum_type service; rdm_device_enum_type dev);

void rdm_data_got_atbrew(void);
```

## File: rdevmap.c

### New functions before rdm_init() function

Line:309

```
sio_port_id_type
   rdm_register_new_service( rdm_device_enum_type dev,
                    rdm_service_open_func_ptr_type  open_fn,
                    rdm_service_close_func_ptr_type close_fn,
                    rdm_service_enum_type          *pService)
{

   int service;

   if(!open_fn || !close_fn || dev <= RDM_NULL_DEV || dev >= RDM_DEV_MAX) {
      return SIO_PORT_NULL; //wrong arguments
   }

   //Make sure the requested device is compiled in and available...
   if(rdm_device_to_port_id_table[dev] == SIO_PORT_NULL) {
      return SIO_PORT_NULL; //wrong arguments
   }


   //first find an open service...
   for(service = RDM_DYNAMIC_SRVC1; service <= RDM_DYNAMIC_SRVC_LAST; service++) {
      if(rdm_service_open_routines[service] == NULL)
         break;
   }

   if( service > RDM_DYNAMIC_SRVC_LAST) {

      return SIO_PORT_NULL; //failed, no free ports...

}

   rdm_service_open_routines[service]  = open_fn;
   rdm_service_close_routines[service] = close_fn;

   rdm_configuration_table[service][dev] = dev;


   *pService = service;
   return rdm_device_to_port_id_table[dev];
}


void rdm_unregister_service(rdm_service_enum_type service, rdm_device_enum_type dev)
```

```
{
   if(service <= RDM_NULL_SRVC || service >= RDM_SRVC_MAX) {
      return;
   }

   rdm_service_open_routines[service]  = NULL;
   rdm_service_close_routines[service] = NULL;

   rdm_configuration_table[service][dev] = RDM_SRVC_NOT_ALLOWED;
}


void rdm_init_config_table(void)
{
   int service;
     for(service=RDM_DIAG_SRVC; service < RDM_SRVC_MAX; service++)
  {
     int dev = RDM_NULL_DEV;
     rdm_configuration_table[service][dev]=RDM_NULL_DEV;
     dev++;
     for( ; dev < RDM_DEV_MAX; dev++)
     {
        rdm_configuration_table[service][dev]=RDM_SRVC_NOT_ALLOWED;
     }
  }
}


void rdm_init_current_config_table(void)
{
  int i;

  //first set the defaults...
  for(i=RDM_DIAG_SRVC; i < RDM_SRVC_MAX; i++)
  {
     rdm_current_config_table[i] = RDM_NULL_DEV;
  }

}
```

## In function rdm_init()

Line:477

*Before:*

```
rdm_menu_init();
```

*After:*

```
rdm_menu_init();
rdm_init_current_config_table();
```

### Replace the rdm_send_service_cmd() with the following code

```
LOCAL boolean rdm_send_service_cmd
(
  rdm_service_enum_type  service,
  rdm_command_enum_type  port_cmd,
  rdm_device_enum_type   device
)
{

  sio_port_id_type sio_port = rdm_device_to_port_id_table[device];

#ifdef RDM_DEBUG
#error code not present
#endif

  if((service <= RDM_NULL_SRVC) ||
     (service >= RDM_SRVC_MAX))
  {
     ERR_FATAL( "Invalid Service Task type", 0, 0, 0);
  }

  if(rdm_service_open_routines[service] == NULL ||
     rdm_service_close_routines[service] == NULL) {
     return FALSE;
  }

  if(port_cmd == RDM_OPEN_PORT)
  {
     (rdm_service_open_routines[service])(sio_port);
  }
  else if(port_cmd == RDM_CLOSE_PORT)
  {
     (rdm_service_close_routines[service])();
  }
  else
  {
     ERR_FATAL( "Invalid Service Task type", 0, 0, 0);
  }

  return(TRUE);

} /* rdm_send_service_cmd */
```

### Replace rdm_register_close_func() with the following code:

```
void rdm_register_close_func
(
  rdm_service_enum_type               service,
  rdm_service_close_func_ptr_type     close_func
)
{
```

```
#ifdef RDM_DEBUG
#error code not present
#endif

  if((service <= RDM_NULL_SRVC) ||
  (service >= RDM_SRVC_MAX))
  {
     ERR_FATAL( "Invalid Service Task type", 0, 0, 0);
  }

  rdm_service_close_routines[service] = close_func;

} /* rdm_register_close_func() */
```

Replace rdm_register_open_func() with the following code:

```
void rdm_register_open_func
(
  rdm_service_enum_type             service,
  rdm_service_open_func_ptr_type    open_func
)
{

#ifdef RDM_DEBUG
#error code not present
#endif

  if((service <= RDM_NULL_SRVC) ||
  (service >= RDM_SRVC_MAX))
  {
     ERR_FATAL( "Invalid Service Task type", 0, 0, 0);
  }

  rdm_service_open_routines[service] = open_func;
} /* rdm_register_open_func() */


void rdm_data_got_atbrew(void)
{
  int service;

#ifdef RDM_DEBUG
#error code not present
#endif

  rdm_device_enum_type dev = rdm_get_device(RDM_DATA_SRVC);

      //first find an open service...
   for(service = RDM_DYNAMIC_SRVC1; service <= RDM_DYNAMIC_SRVC_LAST; service++)
   {
      if(rdm_configuration_table[service][dev] == dev)
        break;
   }

   if(service <= RDM_DYNAMIC_SRVC_LAST)
   {
      rdm_assign_port(service, dev, NULL);
   }
} /* rdm_data_got_atbrew() */
```

In function rdm_set_bt_mode

Line 1877

*After the first INTLOCK(), add the following line of code:*

```
rdm_init_config_table();
```

# Configuring Devices

Before BREW can operate properly, you must implement the configuration functions listed in this section.

## Physical and hardware characteristics

The following functions are used to retrieve the current handsets' physical and hardware characteristics:

- OEM_GetDeviceInfo()
- OEM_GetDeviceInfoEx()
- OEM_GetConfig()

AEEDeviceItem is the data type passed to OEM_GetDeviceInfoEx(). OEMs must implement support for all the defined device items, which are described in the BREW API Reference guide under the AEEDeviceItem datatype. The items are defined are in the file AEEShell.h.

AEEConfigItem is the data type passed to OEM_GetConfig. OEMs must implement support for all the defined config items, which are described in the BREW OEM API Reference under the OEM_GetConfig and OEM_SetConfig functions.

Some device and config item support is provided in the files OEMConfig.c and OEMSVC.c. OEMs must complete the implementation for their device where indicated, and review the remaining implemenations to verify they are suitable for their device.

### BREW heap

The function, OEM_GetinitHeapBytes, is used to configure the size and location of the BREW supervisor and user mode heaps. The Brew heap is used by all BREW applications, and the size and number of BREW applications, which can execute simultaneously on a device, is dictated by the amount of heap made available to BREW.

Certain BREW extensions may have certain memory requirements so it is recommended that you make the largest possible memory available to BREW. If BRIDLE is enabled on the handset build, you must dedicate an additional supervisor heap, which is used for memory allocations of the BREW code for supervisor mode or BREW Extensions. At a minimum, this supervisor heap must be 64 KB. The actual size needed may be more than this if the ICAMERA interface is implemented or if any static extensions that require supervisor heap allocations are implemented.

## Download services parameters

The following functions are used to retrieve/set the configuration parameters related to the download services:

- OEM_GetConfig()
- OEM_SetConfig()

BREW applications with system privileges can retrieve/set the configuration parameters using the ICONFIG interface.

## Configuring R-UIM-based devices

For R-UIM-based devices, the configuration parameters must come from either the ME (Mobile Entity) or the R-UIM. If you are working with an operator who requires an R-UIM, you must comply with the following division of the sources from which information is obtained. The sources are either ME (defined as the R-UIM-based mobile station without an R-UIM), or R-UIM.

| **Configuration Parameter** | **ME** | **R-UIM** |
|---|---|---|
| CFGI_ALLOW_3G_2G_FAILOVER | X | |
| CFGI_APP_KEY_1 | X | |
| CFGI_APPX5_MPC_ADDR | X | |
| CFGI_APPX5_MPC_PORT | X | |
| CFGI_AUTOSTART | X | |
| CFGI_AUTOSTARTARGS | X | |
| CFGI_BUSY_CURSOR_OFFSET | X | |

| | | |
|---|---|---|
| CFGI_CACHED_RESOURCES | X | |
| CFGI_CARDID_LEN | | X |
| CFGI_CARDID | | X |
| CFGI_CLOSE_KEYS | X | |
| CFGI_DEBUG_KEY | X | |
| CFGI_DISALLOW_DORMANCY | X | |
| CFGI_DNS_IP1 | X | |
| CFGI_DNS_IP2 | X | |
| CFGI_DORMANCY_NO_SOCKETS | X | |
| CFGI_DOWNLOAD : dwCarrierID | X | |
| CFGI_DOWNLOAD : dwPlatformID | X | |
| CFGI_DOWNLOAD : bBKey | X | |
| CFGI_DOWNLOAD : bAKey | | X |
| CFGI_DOWNLOAD : szServer | X | |
| CFGI_DOWNLOAD : wFlags | X | |
| CFGI_DOWNLOAD : nAuth | X | |
| CFGI_DOWNLOAD : nPolicy | X | |
| CFGI_DOWNLOAD_BUFFER | X | |
| CFGI_DOWNLOAD_FS_INFO | X | |
| CFGI_FILE_CACHE_INFO | X | |
| CFGI_GPSONE_LOCK | X | |
| CFGI_GPSONE_SVRIP | X | |
| CFGI_GPSONE_SVRPORT | X | |
| CFGI_GPSONE_TRANSPORT | X | |
| CFGI_GSM1X_IDENTITY_PARAMS | | X |
| CFGI_GSM1X_PRL | | X |

| **Configuration Parameter** | **ME** | **R-UIM** |
|---|---|---|
| GFGI_GSM1X_RTRE_CONFIG | | X |
| CFGI_GSM1X_SID_NID_PARAMS | | X |
| CFGI_HTTP_BUFFER | X | |
| CFGI_ISTATIC_SCROLL | X | |
| CFGI_KB_AUTOREPEAT | X | |
| CFGI_MAX_DEMO_ITEMS | X | |
| CFGI_MAX_DISPATCH_TIME | X | |
| CFGI_MIN_IDLE_TIME | X | |
| CFGI_MODULE_FSLIMIT | X | |

| | | |
|---|---|---|
| CFGI_MOBILEINFO : nCurrNAM | | X |
| CFGI_MOBILEINFO : dwESN | X | |
| CFGI_MOBILEINFO_szMobileID | | X |
| CFGI_NET_CONNTIMEOUT | X | |
| CFGI_OFFLINE_PPP_TIMEOUT | X | |
| CFGI_SAFEMODE_STARTMODE | X | |
| CFGI_SAFEMODE_TIMER | X | |
| CFGI_SCREEN_SAVER | X | |
| CFGI_SLEEP_TIMER_RESOLUTION | X | |
| CFGI_SUBSCRIBERID | | X |
| CFGI_SUBSCRIBERID_LEN | | X |
| CFGI_SYSMEM_SIZE | X | |
| CFGI_WEB_IDLECONNTIMEOUT | X | |

Before you call AEE_Init(), make sure that the R-UIM initialization completes and that the valid SID can be obtained from the R-UIM. You can verify this by ensuring that OEM_GetConfig() for CFGI_SUBSCRIBERID, CFGI_SUBSCRIBERID_LEN, CFGI_CARDID and CFGI_CARDID_LEN returns correct values.

If a SID changes after BREW is initialized, BREW must be notified of the new value by means of the IDOWNLOAD_SetSubscriberID() function. Refer to the *BREW SDK API Reference Online Help* for information.

Following is sample code that you can use when the SID changes:

```
int SetSubscriberID(const char * pszSID, int nLen)
{
   int nErr;
   IShell * pIShell = AEE_GetShell();
   IDownload * pIDownload = NULL;

   if (!pIShell) return EFAILED;
   nErr = ISHELL_CreateInstance(pIShell, AEECLSID_DOWNLOAD, (void **)
&pIDownload);
   if (nErr == SUCCESS && pIDownload)
   {
      IDOWNLOAD_SetSubscriberID(pIDownload, pszSID, nLen);
      IDOWNLOAD_Release(pIDownload);
   }
   return nErr;
}
```

## R-UIM interface

The R-UIM interface is a collection of functions with the following capabilities:

- Verifies the R-UIM card connection.
- Returns the current R-UIM status.
- Compares the designated card holder verification (CHV) on the R-UIM with the PIN passed.
- Changes the designated CHV on the R-UIM to the PIN passed.

## Overview of a R-UIM-based device

In non-R-UIM-based systems, the ESN is used for authentication of mobile stations through the CAVE algorithm. R-UIM-based mobile stations can use either the ME ESN or R-UIM ID for CAVE.

The contents of an R-UIM are organized into Dedicated Files (DF) and Elementary Files (EF). There are three relevant EFs:

- EF 6F38 (ESN_ME): Where the ME's ESN is stored. The ME transfers its ESN to this EF when the ME detects that a new R-UIM is inserted into the ME.
- EF 6F31 (user identity module ID [UIMID]): The ID of the R-UIM.
- EF 6F42 (UIMID indicator): Dictates whether ESN_ME or the UIMID is used for the CAVE. The value 0 indicates ESN_ME is used, and the value 1 indicates UIMID is used.

Some carriers require that UIMID be used for authentication, however, for the purpose of BREW porting, ESN_ME must be used for any ESN-related operations (for example, in the structure returned through OEM_GetConfig() for the item CFGI_DOWNLOAD).

## Overview of BREW on a R-UIM-based device

To prevent illegal copying of BREW applications from one device to another, BREW encrypts the application files when they are downloaded. Some of the device-specific characteristics, such as the ESN, are used while encrypting the files. BREW allows only valid and properly encrypted applications devices. When applications are moved to another device or if the characteristics of the device, such as the ESN, change when the R-UIM card is swapped, the applications are declared invalid and deleted from the device.

BREW relies on the fact that the device's ESN does not change when the R-UIM card is changed or swapped. The Subscriber ID (SID) is allowed to change when the R-UIM Card is swapped.

## Porting BREW on R-UIM devices

You must follow these steps to port BREW on a R-UIM-based device:

1. Verify that the OEMRUIM.c file is included in the device build. All functions defined in OEMRUIM.c must be supported.

   **NOTE:** The reference implementation provided with the Porting Kit for MSM Platforms provides an implementation for all functions.

2. Verify that all address book functions are supported—the IAddrBook interface must be supported for accessing AddrBook on a R-UIM device. To create an instance of IAddrBook to access the address book on a R-UIM device, the ClassID, AEECLSID_ADDRBOOK_RUIM, will be used during ISHELL_CreateInstance().

   **NOTE:** The reference file OEMAddrBookRUIM.c provides the implementation for access to the address book on the R-UIM card.

3. Ensure that the flag DIF_SID_ENCODE is not set. This means that, when OEM_GetConfig() is invoked for the item CFGI_DOWNLOAD, you must ensure that the flag DIF_SID_ENCODE is *not* set in the wFlags member of the AEEDownloadInfo structure.

4. Ensure that a valid ME ESN is returned inside OEM_GetConfig() for CFGI_MOBILE_INFO. The following are the ME ESN requirements:
   - The dwESN member of the AEEMobileInfo structure must be a valid ME ESN and must uniquely identify this device. No two devices share the same ESN.
   - The ME ESN must be kept constant and must NOT change when the R-UIM card is swapped.
   - The ME ESN is also used by application developers while generating test signatures for the device. Hence, you must allow application developers to obtain the ESN of a device.
5. By default, BREW allows all non-subscription applications to be used even when the R-UIM card originally used to download the applications is swapped or changed. Subscription-based applications are used only by the original user who downloaded the applications. To override this behavior and ensure that all applications can be used only by the original user who downloaded them, set the flag DIF_SID_VALIDATE_ALL inside the wFlags member of the AEEDownloadInfo structure when OEM_GetConfig() for CFGI_DOWNLOAD is invoked.

   **NOTE:** Setting this flag prevents BREW applications from being executed when the R-UIM card originally used to download the applications is changed.
6. By default, BREW allows deletion of BREW applications only by the owner who downloaded the applications. To override this behavior and allow one R-UIM user to delete applications owned by another, set the flag DIF_RUIM_DEL_OVERRIDE inside the wFlags member of the AEEDownloadInfo structure when OEM_GetConfig() for CFGI_DOWNLOAD is invoked.
7. Allow the IAddrBook interface to access the address book on the R-UIM card. BREW applications use the ClassID AEECLSID_ADDRBOOK_RUIM to create an instance of the IAddrBook interface; this allows the application to access the AddressBook on the R-UIM card.
8. Provide a menu item which shows the mobile station's ME ESN. This item is critical to developers, who need to create test signatures based on the ME ESN.
9. If an ME ESN is not required by the carrier and cannot be assigned to each device, you must ensure that the dwESN member of the AEEMobileInfo structure gets assigned a unique 32-bit unsigned integer in the intended carrier's network.

   **NOTE:** After it is assigned in the factory, this number remains unchanged throughout the life span of the mobile station. This could be a unique serial number assigned by the manufacturer.

## Verifying implementation

The following are the set of tests you must perform to ensure the successful porting of BREW on a R-UIM-based device.

### To run test 1

1. Download subscription and non-subscription BREW applications on a R-UIM-based device.
2. Change the R-UIM card on the device.

   *The applications should stay intact and are not deleted.*

   *If the DIF_RUIM_DEL_OVERRIDE flag is set, you can delete applications; if not set, you cannot delete applications.*

   *If the DIF_SID_VALIDATE_ALL flag is set, you cannot run any application (subscription or non-subscription); if not set, you can run only non-subscription applications.*

### To run test 2

1. Download BREW applications on a R-UIM-based device.
2. Change the R-UIM card and power-cycle the device.
3. Replace the original card.

*You should be able to execute each of the BREW applications and also delete each one of them.*

### To run test 3

In a BREW application, create an instance of the IAddrBook interface using the classID AEECLSID_ADDRBOOK_RUIM.

*The application should be able to access the address book on the R-UIM card.*

**NOTE:** For all China Unicom phones, ensure that the DIF_SID_VALIDATE_ALL and DIF_RUIM_DEL_OVERRIDE flags are set.

## Maximum path length and mapping

The following functions are used to determine the maximum path length and the mapping between BREW and native directory names:

- OEMFS_GetMaxPathLength()
- OEMFS_GetNativePath()
- OEMFS_GetBREWPath()

## Appearance

To configure the appearance of UI items, you can implement the optional function OEM_GetItemStyle():.

See the *BREW™ OEM API Reference Online Help for MSM Platforms* for more information on the functions.

## Packet Data Dormancy

Before version 3.0.3, BREW could initiate mobile packet data dormancy, depending on OEM configuration. In BREW 3.0.3, BREW's mobile initiated dormancy code was removed, which means the CFGI_DISALLOW_DORMANCY and CFGI DORMANCY _NO_SOCKETS configuration items no longer exist. Even though BREW does not initiate packet data dormancy, it still supports it (applicable to CDMA 1x). Lower layers, such as the AMSS, are responsible for managing and initiating dormancy.

BREW supplies the INETMGR_SetDormancyTimeout() API for setting the dormancy time-out value which is the amount of time after which mobile initiated dormancy should be initiated, if during this whole time the packet data call was idle. It is propagated to lower layers through OEMNet_SetDormancyTimeout().

In BREW 3.0.3 and later, OEMs that disabled BREW initiated dormancy by setting CFGI_DISALLOW_DORMANCY to TRUE, can achieve the same behavior by using an empty implementation of OEMNet_SetDormancyTimeout() that just returns DSS_SUCCESS, without changing the packet data inactivity timer value.

A new CFGI_DORMANCY_TIMEOUT_DEFAULT configuration item may be used to set a default dormancy time out value for the device. IN BREW 3.0.3 and later, OEMs that used to set CFGI_DORMANCY_NO_SOCKETS to TRUE, can achieve similar behavior by setting the default linger value to the max value, while setting the default dormancy time out value to a normal value, such as the default 30 seconds.

## BREW file access restrictions

The purpose of this feature is to impose restrictions on remotely accessing certain types of files on a device. Remote access refers to accessing files on a device through the serial interface using diagnostics, such as EFS Explorer.

This section discusses the steps to enable the access restrictions and the architecture of the remote file access restrictions. It describes how to discover whether or not your DMSS builds contain these features, and, if so, how to incorporate them.

### Architecture

The following are required:

1. Registration function OEMFS_RegRmtAccessChk() that can be invoked by BREW.
2. Invoking the AEE layer callback function when a remote file access request is made to a file contained in one of the registered directories.
3. Blocking or allowing file access depending on the response of the callback.

The first requirement is met by providing the following function to the AEE layer:

```
void OEMFS_RegRmtAccessChk( const char **pszDirList,
uint32 nListElements,
PFNCHKRMTACCESS pfn )
```

This function has been provided for you in OEMFS.c; no modifications are necessary. This function is called during device initialization by the AEE layer and is used to register a list of directories to monitor, including the AEE layer callback function to be called when a remote file access request is made.

The second requirement is met by calling the AEE layer callback when a file access is requested. This means calling the function pointed to by the argument pfn of OEMFS_RegRmtAccessChk() when an access request is made to a file contained in one of the directories listed in the OEMFS_RegRmtAccessChk()ís pszDirList argument. The AEE layer callback returns either TRUE or FALSE, signaling whether file access is allowed or denied. A reference implementation for this is already provided in OEMFS.c. No modifications are necessary.

The third requirement is met by blocking or allowing the file access based on the return value from the callback function of the second requirement. The DMSS files provide this implementation.

## Porting instructions

### Step 1. Verify whether or not you have the DMSS patch to support this feature

This feature requires a patch to the DMSS. The access restriction and memory operations features are not included in all DMSS builds. Verify if your build contains them; if not, integrate them using the following information.

The access restriction feature is based on the feature definition: FEATURE_DIAG_FS_ACCESS_VALIDATION. The memory operations feature is based on the feature definition: FEATURE_DIAG_DISALLOW_MEM_OPS.

To determine if a build contains both of these mandatory features, search custsurf.h, custefs.h, and custdiag.h. Both of these feature definitions must be defined. A successful search for the DMSS definitions described above indicates that the build contains these features. If you find that the build does not include these features, obtain the DMSS patch by contacting brew-oem-support.

### Step 2. Enable the feature in the device build.

After obtaining the DMSS patch, define the following macros while doing the device build:

- FEATURE_DIAG_DIS
- ALLOW_MEM_OPS

Do not make any modifications to the file OEMFS.c (particularly, the sections covered by the above feature definitions).

If you are using the 5100 series MSM, see Special instructions for 5100 series MSM on page 74.

### Step 3. Testing the feature

After completing the device build, perform the following tests to ensure that it has been incorporated correctly.

**NOTE:** These tests will be included in the PEK in a later release.

#### To test the device build

1. Using a tool such as EFS Explorer or AppLoader, ensure that the following types of files cannot be copied from the device:
   - prefs.dat from the brew directory
   - Any numbered MIF (for example, 450.mif) inside the brew directory
   - Any file from a directory inside the brew directory that is all numbered (for example, 450)
   - Any file from the download directory within the brew directory
2. Ensure that the following file types can be copied from the device:
   - Any file from the shared directory
   - A named MIF (for example, hello.mif) inside the brew directory
   - Any file from a directory that is named (for example, hello) inside the brew directory

## Special instructions for 5100 series MSM

OEMs must make the following three DMSS changes even if the version of DMSS they are using includes the remote file access restrictions.

### First change

From inside the function diag_fs_read(), remove the line:

```
DIAG_FS_VALIDATE_ACCESS( READ, req_ptr->filename_info.name );
```

from its current location, and move it to the location shown below (at approximately at Line 655 of the same function):

```
/*--------------------------------
Check for valid packet length.
--------------------------------*/
expected_pkt_len =
sizeof(req_ptr->seq_num) +
sizeof(req_ptr->filename_info.len) +
req_ptr->filename_info.len;
if (pkt_len != expected_pkt_len)
{
return (ERR_PKT(DIAG_BAD_LEN_F));
}
```

**MOVE LINE TO HERE**

```
}
else if (next_seq_num == req_ptr->seq_num)
{
/*--------------------------------
Check for valid packet length.
--------------------------------*/
expected_pkt_len = sizeof(req_ptr->seq_num);
```

## Second change

From inside the function diag_fs_write(), remove the line:

```
DIAG_FS_VALIDATE_ACCESS (WRITE,
req_ptr>block.begin.var_len_buf.name_info.name);
```

from its current location, and move it to the location shown below (approximately at Line 894 of the same function):

```
if (pkt_len != expected_pkt_len)
 {
 next_seq_num = 0;
 return (ERR_PKT(DIAG_BAD_LEN_F));
 }
```

**MOVE LINE TO HERE**

```
} /* Sequence number == 0 */
 else if (next_seq_num == req_ptr->seq_num)
 {
 /*----------------------------------------
 Assign data_ptr to request data block
 ----------------------------------------*/
 data_ptr = &req_ptr->block.subseq;
```

## Third change

From inside the function diag_fs_iter(), remove the line:

```
DIAG_FS_VALIDATE_ACCESS( ITERATE, req_ptr->dir_name.name );
```

from its current location, and move it to the location shown below (at approximately Line 1274 of the same function):

```
if (fs_rsp.enum_init.status != FS_OKAY_S)
 {
 rsp_ptr = (diag_rsp_type *) diagbuf_pkt_alloc(rsp_len);
 break;
 }
```

**MOVE CODE TO HERE**

```
} /* if first time */
 /*-----------------------------------------------------------------------
 Request the directory name corresponding to the given sequence number
 -----------------------------------------------------------------------*/
 fs_enum_next(&fs_enum_data,
 &fs_enum,
 NULL,
 &fs_rsp);
```

# Managing and Downloading Applications and Extensions

This section provides an overview of BREW file types and discusses all aspects of BREW application management and download from an OEM's perspective.

## BREW file types and dynamic application installations

The following basic file types are specific to BREW:

- MIF: Contains information about each module. Each module can contain one or more BREW applications, and this type of file contains the following information:
  - Applet ClassIDs, names, images, and settings
  - General module information in text
  - Privilege levels
  - Exported ClassIDs and MIME types
  - Module dependencies

  **NOTE:** The MIF filename needs to match the application directory name.

- Dynamically loaded module (MOD file): The dynamically loaded module that is executed at runtime. The applet source files are compiled and linked into this MOD file type.
- BAR file: Contains all the application resources; that is, text strings, images and dialogs. This file type is used by the MOD files during execution.
- Signature (SIG) file: Contains digital signature information to ensure the integrity of each applet. When a module is downloaded, this signature is verified by validating the binary and resource files. This mechanism is also exercised in the beginning of each execution. The SIG filename must match the MOD filename. For example, if the MOD file is called myapp.mod, the SIG filename must be myapp.sig. There are three types of SIG files:

– Test: Used only during development of applets or devices. This file is  generated based on each device's 32-bit ESN.

– Each device needs a different SIG file, but all the applets within the same device can share the same signature. You can get this signature file directly from QUALCOMM.

– Carrier pass-through: Generated if an operator certifies an application. This file is generated based on the contents of the module.

– TRUE BREW®: Generated if the application goes throughTRUE BREW testing. This file is generated based on the contents of the module.

Filenames in the canonical BREW file name space are:

fs:/

The remaining filename space scheme has the following key features:

- Backward compatibility
- New filenames distinct from the current name space
- An unambiguous, complete name space

The application relative name space is now:

fs:/~/

BREW maps fs:/~/ to fs:/mod/<current app directory>". The fs/~/ structure is designed as a convenience for application developers by allowing them to address a file in their own directory without requiring them to know their own module ID or construct a special path that includes it. This name space also allows applications to publish files for sharing with other applications without being concerned about name space conflicts.

Some concrete example paths and their meanings are as follows:

| | |
|---|---|
| fs:/~/ | The current application's module directory. If no application is active, this means fs:/sys/. |
| fs:/ | The root directory from a BREW perspective. Any application can initialize an enumerator on this directory, but the enumeration results depend on the current application's permissions. |

| | |
|---|---|
| fs:/~0x0100F00D/ | The module directory of the module that exports the class 0x0100F00D. **NOTE:** In case 0x0100F00D is exported by module 333, this path is rewritten by BREW to fs:/mod/333/. |
| fs:/mod/ | The modules directory, currently considered the application directory in the SDK. |
| fs:/mif/ | The MIF directory. |
| fs:/shared/ | What BREW expects the OEM file system to map to the shared directory. |
| fs:/ringers/ | What BREW expects the OEM file system to map to the ringers directory. |
| fs:/address/ | What BREW expects the OEM file system to map to the address book directory. |
| fs:/card0/ | What BREW expects to be the first removable memory card. |

**NOTE:** Anything that doesn't begin with fs:/ is treated as a case-insensitive pathname and is converted to lowercase. The converted string is appended to the new filename space under fs:/~/.

The AEEFile.h contains constant strings (shown below) to conveniently access the new filename space. These strings reflect the manner in which BREW expects to use the file system.

```
#define AEEFS_ROOT_DIR          fs:/
#define AEEFS_HOME_DIR          fs:/~/
#define AEEFS_SYS_DIR           fs:/sys/
#define AEEFS_MOD_DIR           fs:/mod/
#define AEEFS_MIF_DIR           fs:/mif/
#define AEEFS_SHARED_DIR        fs:/shared/
#define AEEFS_ADDRESS_DIR       fs:/address/
#define AEEFS_RINGERS_DIR       fs:/ringers/
#define AEEFS_CARD0_DIR         fs:/card0/
```

You are responsible for mapping BREW name space names in the  OEMFSPath.c and OEMFS.c to their proper locations on your native file system.

When mapping the BREW name space names to the native file system, please consider the following requirements:

- AEEFS_SYS_DIR must not map to any type of removable media and must not be the same as or any subdirectory of AEEFS_MOD_DIR, AEEFS_SHARED_DIR, AEEFS_ADDRESS_DIR or AEEFS_RINGERS_DIR.
- AEEFS_MOD_DIR must not map to any type of removable media and must not be the same as or any subdirectory of AEEFS_SHARED_DIR, AEEFS_ADDRESS_DIR or AEEFS_RINGERS_DIR. It is also recommended that AEEFS_MOD_DIR does not map to the same area as AEEFS_MIF_DIR.
- AEEFS_MIF_DIR must not map to any type of removable media and must not be the same as or any subdirectory of AEEFS_SHARED_DIR, AEEFS_ADDRESS_DIR or AEEFS_RINGERS_DIR. It is also recommended that AEEFS_MIF_DIR does not map to the same area as AEEFS_MOD_DIR.
- AEEFS_SHARED_DIR must not map to any type of removable media and must not be the same as or any subdirectory of AEEFS_ADDRESS_DIR or AEEFS_RINGERS_DIR.
- AEEFS_ADDRESS_DIR must not be the same as or any subdirectory of AEEFS_SHARED_DIR or AEEFS_RINGERS_DIR.
- AEEFS_RINGERS_DIR must not map to any type of removable media and must not be the same as or any subdirectory of AEEFS_ADDRESS_DIR or AEEFS_SHARED_DIR.

BREW provides mapping from old names to new names and from fs:/~/ names to their canonical form.

| | |
|---|---|
| fs:/ maps to brew:/ | This means that fs:/mod/ and fs:/mif/ are mapped to brew/mod/ and brew/mif/, respectively.<br>**NOTE:** If you have any dynamic applications on your phone, you'll have to move them. |
| fs:/card0/ maps to mmc1/ on platforms that define FEATURE_MMC | This is an exception to the previous rule regarding fs:/ mapping to brew/. |

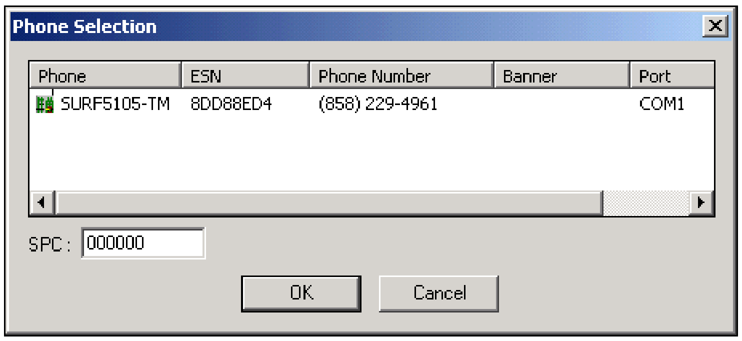
## Installing BREW files

You can install BREW-specific files by using the QCT EFS Explorer.

## To install the BREW files

1. Open the EFS Explorer and make sure the device is in Diagnostic Monitor (DM) mode.

   *The Phone Selection dialog box opens.*



| Phone | ESN | Phone Number | Banner | Port |
|---|---|---|---|---|
| SURF5105-TM | 8DD88ED4 | (858) 229-4961 | | COM1 |

SPC : 000000

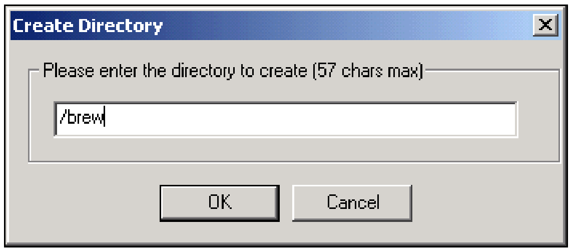
2. Select the device.

   *The selected device is highlighted.*

3. In the service programming code (SPC) field, enter the correct SPC and click **OK**.

4. Go to the BREW root directory. If it doesn't exist, create one by clicking **File** > **New** > **Directory**.

   *The Create Directory dialog box opens.*

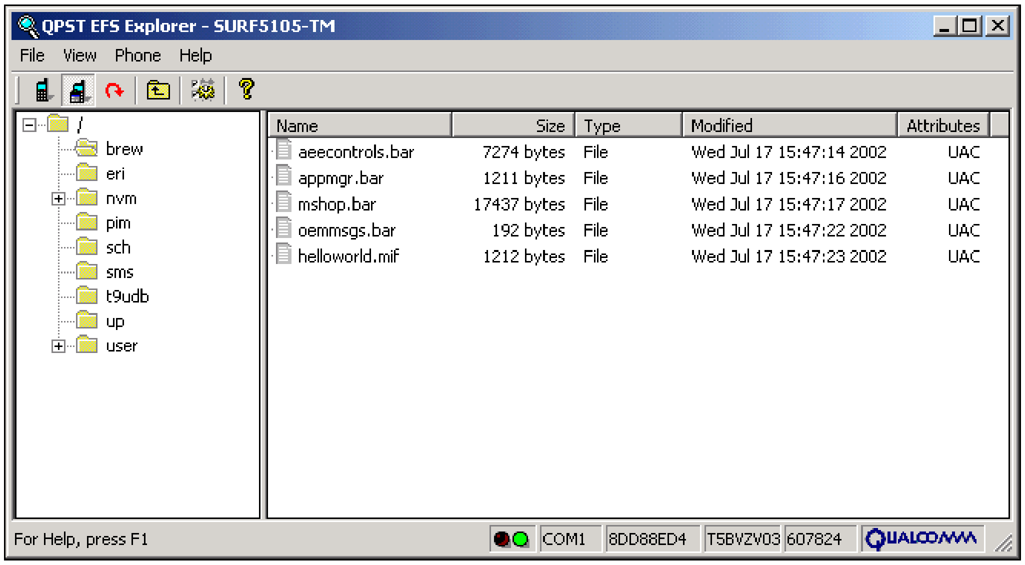


Please enter the directory to create (57 chars max)

/brew

5. Copy the MIFs to the root directory by clicking **File** > **New** > **File** or by dragging the MIF over the EFS Explorer window.

   *The MIFs are moved to the BREW directory.*



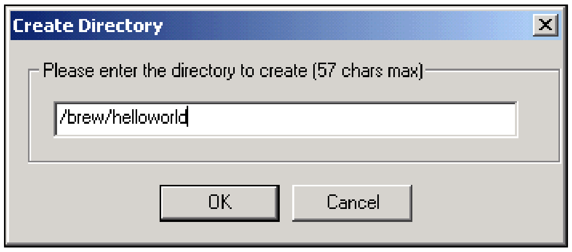
QPST EFS Explorer - SURF5105-TM

| Name | Size | Type | Modified | Attributes |
|---|---|---|---|---|
| aeecontrols.bar | 7274 bytes | File | Wed Jul 17 15:47:14 2002 | UAC |
| appmgr.bar | 1211 bytes | File | Wed Jul 17 15:47:16 2002 | UAC |
| mshop.bar | 17437 bytes | File | Wed Jul 17 15:47:17 2002 | UAC |
| oemmsgs.bar | 192 bytes | File | Wed Jul 17 15:47:22 2002 | UAC |
| helloworld.mif | 1212 bytes | File | Wed Jul 17 15:47:23 2002 | UAC |

For Help, press F1 COM1 8DD88ED4 T5BVZV03 607824 QUALCOMM

**6.** Create the module subdirectory.

*The Create Directory dialog box opens.*



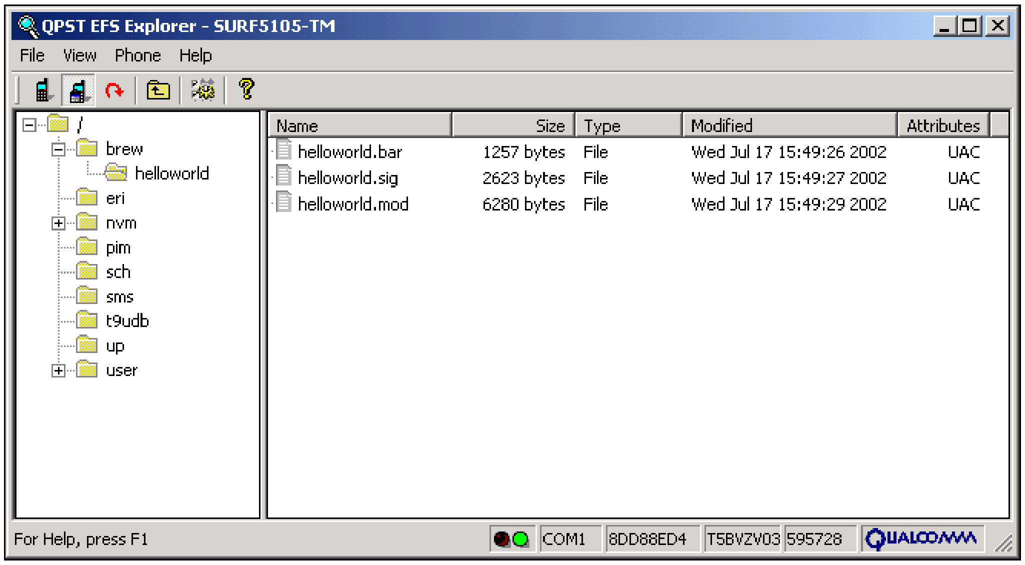
Create Directory

Please enter the directory to create (57 chars max)

/brew/helloworld

**7.** Copy the BAR, MOD, and SIG files to the module subdirectory.

*The BAR, MOD, and SIG files are moved to the module's subdirectory.*



| Name | Size | Type | Modified | Attributes |
|---|---|---|---|---|
| helloworld.bar | 1257 bytes | File | Wed Jul 17 15:49:26 2002 | UAC |
| helloworld.sig | 2623 bytes | File | Wed Jul 17 15:49:27 2002 | UAC |
| helloworld.mod | 6280 bytes | File | Wed Jul 17 15:49:29 2002 | UAC |

For Help, press F1    COM1    8DD88ED4    T5BVZV03 595728

8. Reset the device by clicking **Phone > Reset phone** from the Phone menu.

   *The BREW-specific files are installed.*

## Creating static extension DLLs

This section describes how to create static extension DLLs.

### To create a static extension DLL

1. Create a Win32 DLL.
2. Export a function "const AEEStaticClass* GetStaticClassEntries(void)" from that DLL, which returns an array of AEEStaticClass.
3. Define AEE_STATIC preprocessor define.
4. Build the DLL and be sure it exists in the \Modules directory within the \bin directory where the BREW_Simulator.exe resides.
5. For building the DLL, include paths for AEE header files in your project settings and in your path to BREW_Simulator.lib, which is available in the same directory as BREW_Simulator.exe.

## Sample code for the exported function

The following is sample code for the exported function:

```
const AEEStaticClass gDemoExtensionClasses[] = {
{ AEECLSID_DEMOEXTENSION, 0, 0, 0, DemoExtension_New},
NULL // Always the last one
};

...

const AEEStaticClass* GetStaticClassEntries(void)
{
return gDemoExtensionClasses;
}

...

int DemoExtension_New(IShell *ps, AEECLSID cls, void **ppif)
{
// Add code to implement this function...
}
```

**NOTE:**

- The BREW_Simulator.lib exports all the functions included in AEE_OEM.h. If you refer to any function from AEE_OEM.h, you must link this library in your DLL.
- If you define an entry in AEEStaticClass array within OEMModTableExt.c and create a DLL for the same ClassID, then the dynamic (DLL) one takes precedence.

## Asynchronous BREW Interfaces

This section describes how to write a BREW extension that interacts with the underlying OEM layer in an asynchronous manner.

### System-level service extensions

The BREW layer supports the concept of application-specific data management and callbacks. With BREW applications, it is expected that all callbacks received are in the context of the BREW thread in their application context and on a non-reentrant basis. Although these characteristics serve to ease the burden on application developers who use BREW, it places an extra burden on interfaces that provide access to system-level services. For example, a high-level BREW interface that provides access to a low-level system data acquisition engine is responsible for several issues independent of calls to the underlying system's I/O mechanisms. The following are system-level service extension issues:

- Storage of the current application context. This context (ACONTEXT) is an opaque data object.
- Calls into the underlying system-level API passing interface-specific context data, as appropriate.
- Processing of inter-task/thread callbacks from the system APIs and scheduling of a return to the main BREW thread.
- Dispatching the appropriate interface-specific callback to the calling BREW application with the appropriate application context (ACONTEXT).

BREW provides some system-level APIs that facilitate these mechanisms. When combined with some general guidelines for use, they allow system services to be supported easily with minimal extra overhead. For information on BREW OEM APIs, see the *BREW™ OEM API Reference Online Help for MSM Platforms*.

## Application contexts

BREW maintains the concept of an internal application context. This context applies to the application currently executing. Although you may perceive the active application as the visible application, it is actually independent of this concept because applications may execute from notifications, callbacks, timers, and so forth. To system-level services, the application context is simply an opaque handle. This opaque handle can be passed to some BREW APIs to help facilitate the development of complex APIs that require inter-task communication. You can access to the current application context by using the following function:

```
void * AEE_GetAppContext(void);
```

Call this function to retrieve the application context, so it can be associated with the instance of an interface that has been allocated on behalf of an application. While this is not necessary if the interface is implemented entirely on top of BREW, it may be necessary when system-level services that run under BREW are used for the basis of providing functionality to a BREW application.

## Callbacks

BREW provides a thread-safe, zero-allocation mechanism to schedule callbacks from one thread back in to the BREW task. This mechanism is supported by two interfaces, ISHELL_Resume and ISHELL_ResumeEx.

```
ISHELL_Resume(IShell * pShell, AEECallback * pcb);
AEE_ResumeCallback(AEECallback * pcb, uint16 wFlags, void * pAppContext);
```

ISHELL_Resume is provided for use by BREW applications. It may also be provided by those services that do not require underlying system task support and do not outlive the owner BREW application.

AEE_ResumeCallback is provided for system service use and allows more control over the scheduling of AEECallbacks. Of particular note in this section is the ability to pass the context of an application to the function. This guarantees that the callback is called in the active context of the application context specified. With regard to the AEECallback and resume mechanism, the following items are critical:

- By design, the AEECallback and associated macros function without allocating memory. The specified AEECallback structure is not copied but linked directly into the list of scheduled callbacks. This ensures that the callback will call with no chance for error.
- It is the responsibility of system-level services to specify the proper application context when scheduling callbacks on behalf of an application. Failure to do so causes instability.

The AEE_GetAppContext and AEE_ResumeCallback mechanisms can be combined to speed development of system-level interfaces. The following is an example of what these mechanisms provide:

- An application-level interface to retrieve information about external devices connected to the device; that is, external.
- Notification of 1-N applications that have created instances of the interface when any new device is connected.
- An underlying system driver that provides system-level notification regarding attachment of new devices. The service is provided through some simple system-level APIs and provides notification by way of a callback from another thread.

To develop the system-level interface for the above example, you must:

1. Create an application-level API.
2. Implement the system service.
3. Link the system service to the schedule callbacks to the application.

## Application interface

The first step in creating the service is defining a simple application-level API. The following is a simple example:

```
IDEVICE_Notify(IDevice * po, AEECallback * pcb);

IDEVICE_GetInfo(IDevice * po, DeviceInfo * pd);
```

Where the calling BREW application passes an AEECallback initialized to the callback function, the desired behavior is that this callback is called when a new device is detected. The BREW application then calls IDEVICE_Getinfo() to retrieve information about the new device. It then recalls IDEVICE_Notify to receive notification about the next device. It is important that the notification callback to the application call is in the context of the application. It cannot be called from another thread and must not be called outside the context of the application.

## System service

The system service for this functionality may provide the following APIs:

```
typedef void (*PFNNEWDEVICE)(void * pUser, DeviceInfo * pdi);

void sys_callback_on_new_device(PFNNEWDEVICE pfn, void * pUser);
```

When a new device is detected, the underlying system driver calls the notification callback (PFNNEWDEVICE), passing the specified user context pointer (pUser) and the information on the new device.

## Sample implementation

The sample interface places some unique requirements on the application developer. Specifically, although there may be many instances of the interface created for 1-N applications, only a single instance of the system-level service is necessary. This means that the multiplexing of notifications to the applications must be handled by the developer of the service.

The AEE_GetAppContext and AEE_ResumeCallback functions make these requirements easy to address.

### To develop the sample system level interface

1. Create the application interface:

**a.** Allocate and initialize the IDEVICE interface class.

**b.** Link the pointer to the class into a linked list of classes. This is necessary so that 1-N callbacks can be scheduled when the system service finds a new device.

**c.** Store the creating application context in the class by way of AEE_GetAppContext.

```
Device * gpList = NULL;

int Device_New(IShell * pShell, AEECLSID cls, void ** ppObj)
{
Device * pme;

if(!ppObj)
return(EBADPARM);

// Create the object and init the vtable...

pme = (Device *)MALLOC(sizeof(Device));
if(!pme)
return(ENOMEMORY);
GET_PVTBL(pme, IDevice) = &gDeviceMethods;

// Link it to the list we manage...

pme->m_pNext = gpList;
gpList = pme;

// Store the calling app context...

pme->m_pApp = AEE_GetAppContext();

pme->m_nRefs = 1;
*ppObj = pme;
return(0);
}
```

This function creates the application-level class. It basically allocates memory, initializes data structures, stores the calling applications context, and inserts the class into a list of objects that are iterated when new devices are detected.

**2.** Call the system-service.

```
int Device_Notify(IDevice * po, AEECallback * pcb)
{
Device * pme = (Device *)po;

CALLBACK_Cancel(pme->m_pcbApp); // Cancel any that are pending
pme->m_pAppCallback = pcb;

sys_callback_on_new_device(FoundNewDevice, &gpList);
}
```

This function stores the AEECallback specified for the interface for use when the system-level notification is called. It then calls the system-level driver API, telling it to call the FoundNewDevice function when a new device is connected.

**3.** Handle the system-level callback.

```
static void FoundNewDevice(Device ** ppList, DeviceInfo * pdi)
{
// Copy the data and schedule each of the pending apps

Device * pd;

LOCK_INTERRUPTS();

for(pd = *ppList; pd != NULL; pd = pd->m_pNext){
if(pd->m_pAppCallback){
MEMCPY(&pd->device, pdi, sizeof(DeviceInfo));
AEE_ResumeCallback(pd->m_pAppCallback, 0, pd->m_pApp);
}

UNLOCK_INTERRUPTS();
}
```

This pseudocode is called from the system thread. It iterates the list of instances that have registered to receive notification of a new device and schedules the callback for them in the proper application context.

**NOTE:** This sample code makes the likely invalid assumption that no new devices are registered, and no new devices overwrite the instance of the last found device by the application. However, the point of the sample is not the management of the DeviceInfo structures but of the callbacks.

Although the sample code provided is simplified, it provides an overview of the basic use of the OEM BREW APIs that can be leveraged to develop system-level services that must manage inter-thread notifications. In doing so, it leverages a couple of the BREW mechanisms that reduce the complexity of managing inter-task and inter-application notifications. Use of mechanisms of this sort are an absolute requirement in the development of such services for use in BREW.

## Application downloads

### BREW signature verification

BREW signature verification occurs for dynamic applications when BREW initializes. If the signature verification fails, the applications are removed from the file system.

You can generate test signatures on the BREW web site (https://brewx.qualcomm.com/oem/home.jsp) or by sending a request for a test signature to QUALCOMM. Test signatures are tied to the ESN of the device. These signature (SIG) files can be used across many applications by simply changing the name of the signature. See detailed information in the *BREW SDK™ User Docs*.

## Enabling the BREW Application Manager for OTA download

The Brew Application Manager consists of applications responsible for launch, purchase, presentation and management of applications. For example, Brew AppMgr 2.x consists of AppMgr and MobileShop. OEMs need to enable BrewAppMgr to make the user launch and manage existing applications and purchase and install new applications on the device.

For enabling the BrewAppMgr, OEMs must link the appropriate BrewAppMgr library into the device build, copy the appropriate color depth brewappmgr.mif in the ../brew/mifs folder and copy the appropriate color depth brewappmgr.bar in the .../brew/mods/brewappmgr folder.

**NOTE:** PK 3.0.1 contains BrewAppMgr libraries built for various chipsets. BrewAppMgr libraries provided with the PK are for integration and testing purposes only and are not intended to be used on a commercial device.

BrewAppMgr source code and various carrier-specific customizations of BrewAppMgr are available on the OEM Extranet (https://brewx.qualcomm.com/oem/oemtool.jsp). For a commercial device, OEMs must check with carriers about the choice of BrewAppMgr. Often carriers mandate the use of BrewAppMgr customized according to their business plan, download the carrier's choice of BrewAppMgr and customizations from the OEM Extranet, and integrate these into device builds.

Successful enabling of the BrewAppMgr should result in OEMs being able to see the main menu of the AppMgr and access MobileShop from there.

MobileShop is application used for downloading applications from the Application Download Server (ADS).

BREW uses OEM_GetConfig() with CFGI_DOWNLOAD to access AEEDownloadInfo struct providing ADS configuration information such as server, carrier ID, platform ID, authorization policy, download flags, and privacy policy. OEMs can enable OTA download by configuring AEEDownloadInfo parameters for CFGI_DOWNLOAD in OEM_GetConfig(). See the *BREW OEM API Reference Online Help* for details on OEM_GetConfig(), CFGI_DOWNLOAD and AEEDownloadInfo.

For testing purpose, OEMs can use following configuration of AEEDownloadInfo parameters:

szServer : oemdemo.qualcomm.com
dwCarrierID : 27
dwPlatformID : 600
nAuth : APOLICY_NONE
wFlags : DIF_MIN_FOR_SID
nPolicy : PPOLICY_BREW_OR_CARRIER

Successful enabling of OTA download should result in OEMs being able to download applications using MobileShop.

## Downloading BREW applications

There are two methods of downloading applications to a BREW device.

- Download applications OTA using the BREW download mechanism. This is the mechanism by which end users download BREW applications to the device.
- Download applications directly to the native file system (EFS) without using the BREW download mechanism. This method is intended strictly for testing purposes only and cannot be used by end users.

### Downloading applications directly to the native file system (EFS)

BREW supplies a tool called the BREW AppLoader, which transfers applications to the device. See the *BREW AppLoader User Guide* for details.

## Downloading applications with the BREW download mechanism

### Connecting to an ADS

BREW uses information that you provide in OEM_GetConfig to connect to the ADS. For testing purposes, use the following information for CFGI_DOWNLOAD:

- BREW or carrier signature policy
- Authorization policy none
- Platform ID
- Carrier ID

You must implement the function OEM_SetConfig() when CFGI_DOWNLOAD is passed as the configuration item. When OEM_SetConfig() is called, you must store the download information (AEEDownloadInfo) in persistent storage (for example, NV) and the same information must be retrieved and returned when the function OEM_GetConfig() is called with CFGI_DOWNLOAD. Failing to do so is a fatal error and results in the BREW download mechanism malfunctioning.

When OEM_GetConfig() is called (with CFGI_DOWNLOAD) prior to calling OEM_SetConfig() (with CFGI_DOWNLOAD), you must return your preferred information (OEM default information) for AEEDownloadInfo (for example, PlatformID, ServerAddress, CarrierID, and so on).

The following is some sample code:

```
int OEM_SetConfig(AEEConfigItem i, void * pBuff, int nSize)
{
if(i == CFGI_DOWNLOAD) {
//Store the Information (pBuff) in a persistent Storage (ex: NV)
return(SUCCESS);
}
......
......
}

int OEM_GetConfig(AEEConfigItem i, void * pBuff, int nSize)
{
if(i == CFGI_DOWNLOAD) {
//Retrieve the information (pBuff) stored when OEM_SetConfig() was called.
If no information has been stored,
//retrieve the default information and fill into pBuff and return
return(SUCCESS);
}
......
......
}
```

## Downloading to R-UIM devices

BREW supports the ability to download and manage applications on R-UIM-based devices. However, you download and store applications on the main flash memory of the device. BREW does not support downloading or storing of applications directly on the RUIM card.

You can achieve the following download-related features associated with R-UIM based devices by setting the appropriate flag to AEEDownloadInfo when OEM_GetConfig() is invoked by BREW with CFGI_DOWNLOAD:

- By default, BREW allows the execution of all applications, except the subscription applications, when the card is changed. For example, if you downloaded an application when card A was on the device, the application runs even when you change the card to card B, as long as the application is *not* the subscription license type.

  **NOTE:** To ensure that no applications run, either subscription or non-subscription, set the flag DIF_SID_VALIDATE_ALL in AEEDownloadInfo when you change the card.

- By default, BREW does not delete applications if you change the card. For example, if you download an application using card A, you cannot delete the application if you change to card B. To delete applications even when you change the card, you must set the flag DIF_RUIM_DEL_OVERRIDE in AEEDownloadInfo.
- BREW stores device-specific information in the MIF to prevent the copying of BREW applications from one device to the other. On RUIM-based devices, do not use the DIF_SID_ENCODE flag. Not using DIF_SID_ENCODE instructs BREW to use the ESN to encode the MIFs.

# Memory Security Through BRIDLE

BRIDLE is a mechanism designed to protect BREW and DMSScode and data from BREW applications. BRIDLE uses the services of the Memory Management Unit (MMU) available on MSM chipsets with ARM 9 core, or the Memory Protection Unit (MPU) available on certain MSM chipsets with the ARM7 core. This protection mechanism is being implemented in phases. The first phase of implementation of this protection service is known as BRIDLE-I, and it provides mechanisms to protect all non-BREW code and data on the device from BREW applications.

For more detailed information about BRIDLE, see the BREW *OEM Reference Guide for BRIDLE*.

## BRIDLE-I architecture

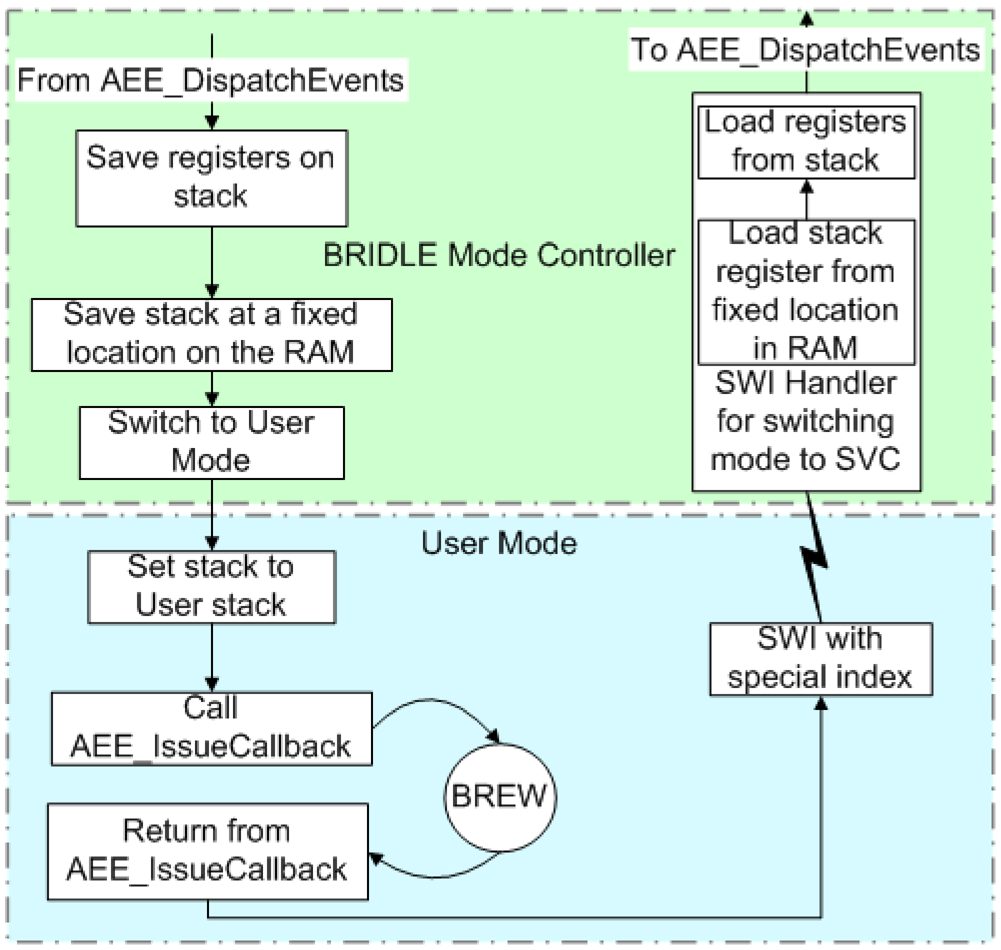
On any mobile device capable of executing third party applications, the code is divided into system software, application service, such as BREW, and user applications. In an unprotected system, the user applications have complete access to the system software, making the device vulnerable to defective or malicious applications. BRIDLE-I protects the system from such applications by restricting access to the system software.

BRIDLE on ARM-based ASICs uses the various modes of operation of the ARM processor core to provide the protection. The seven modes of operation of the ARM processor core are User, Supervisor, System, FIQ, IRQ, Abort, and Undefined. Of these, the DMSS primarily executes in Supervisor, System, FIQ and IRQ modes, and enters Undefined or Abort modes only under abnormal conditions. The DMSS never uses the User mode. As User mode is the most restricted mode of operation, it is the logical choice to run BREW applications. User mode is entered from Supervisor mode by modifying the mode bits of the Current Program Status Register (CPSR). However, the switch from User mode to Supervisor mode can only be done with an interrupt. The only interrupt callable from software is the ARM Software Interrupt (SWI), and BRIDLE uses this as a means for BREW to execute all software that needs to run in Supervisor mode. In BRIDLE-I, the BREW kernel and the underlying operating system execute in Supervisor mode, while BREW applications and facilities, for example, the dispatcher) execute in User mode.

## Supervisor to User

Switching from Supervisor to User mode is safe and relatively simple. On most hardware, it involves setting some bits in a register. In BRIDLE-I, this action is performed during the dispatching of callbacks, so all events are processed in User mode. The next figure illustrates the transition from Supervisor to User mode, and the reverse during function return.

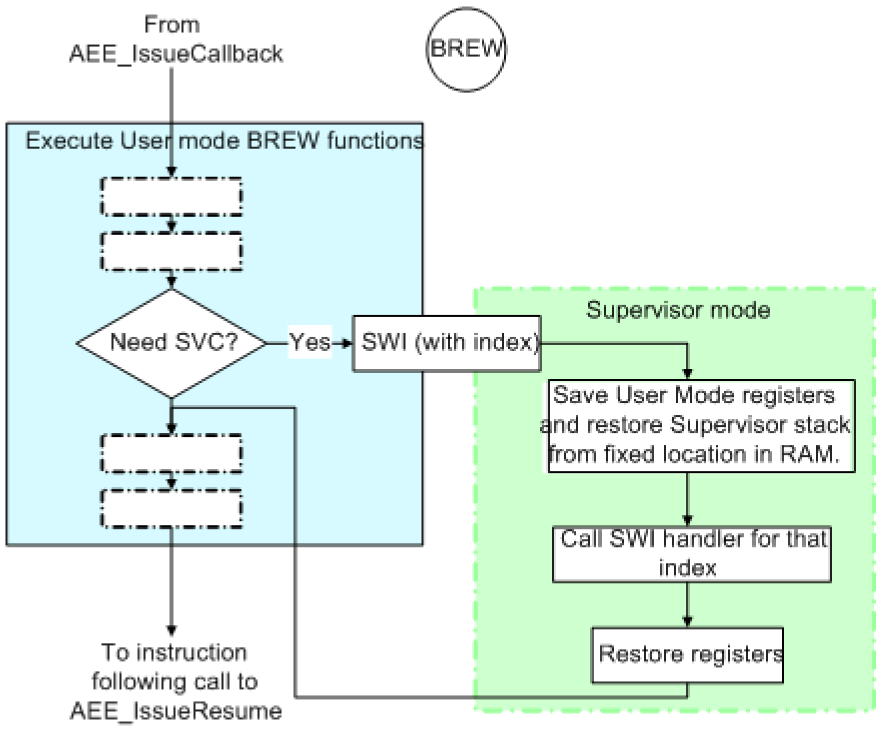
***Supervisor to User mode context switch***



## User to Supervisor mode

Transitioning from User mode to Supervisor mode is more involved. The actual mechanism for this transition involves a SWI in which the handler is installed and executed in Supervisor mode. This may be referred to as a system call and is illustrated in the following figure.

***User to Supervisor mode context switch***


From
AEE_IssueCallback
BREW
Execute User mode BREW functions
Need SVC?
Yes
SWI (with index)
Supervisor mode
Save User Mode registers and restore Supervisor stack from fixed location in RAM.
Call SWI handler for that index
Restore registers
To instruction following call to AEE_IssueResume

## Memory regions

In systems where memory protection is available, a memory access request is resolved, in the case of an MMU-based system, or validated, in the case of an MPU-based system. On ARM-based systems, an illegal access for data causes a data abort, and an illegal access for instructions causes a pre-fetch abort. For the MMU or MPU to control access, the address space is divided up into regions with different access permissions. In BRIDLE-I, there are three distinct memory regions defined as follows.

| **Region** | **Description** |
|---|---|
| Supervisor read/write, User no access | Used for System data (read/write data, zero-initialized data, or bss). |
| Supervisor read/write, User read-only | Used for a small subset of system data that needs to be read in User mode. In particular, the module SWI numbers are required as arguments to the BRIDLE SWI in User mode but are owned by Supervisor mode. |
| Supervisor read/write, User read/write | Used for User data (read/write data, zero-initialized data). |

In BRIDLE-I, the memory regions are established at link time. The rules are supplied to the linker through the ARM scatter load file and establish the coherence between the code and data layout and the memory access control configuration used by the MMU/MPU. The linker uses the object code and the linkage rules and generates an image that can be loaded and executed on the device.

## MMU/MPU configuration

Based on the address space available on the device and the code and data sizes for the various regions, you have to create a configuration for the MMU or MPU before enabling memory protection. This process may take a couple of iterations to optimize code layout, minimize the waste of address space, and, simultaneously, ensure that no code or data is at an address that would compromise BRIDLE security. Also, code or data that should accessible when the processor is in User mode generates data or pre-fetch aborts if such code or data is placed in address regions not accessible in User mode.

A sample MPU configuration is provided in BREWMPU.c, and a sample MMU configuration is provided in ARMMMU.c in the BREW PK. These have to be modified for each OEM's specific architecture. A related function to the MMU/MPU configuration is bridle_checkAccess. This function provides the mechanism for BRIDLE entry points and other BRIDLE support functions to validate accesses from User mode. This function is crucial for BRIDLE to provide protection and is one of the porting aspects validated by OATBridle. A reference implementation for this function is provided in the above-mentioned files, but it may also have to modified and optimized for each device.

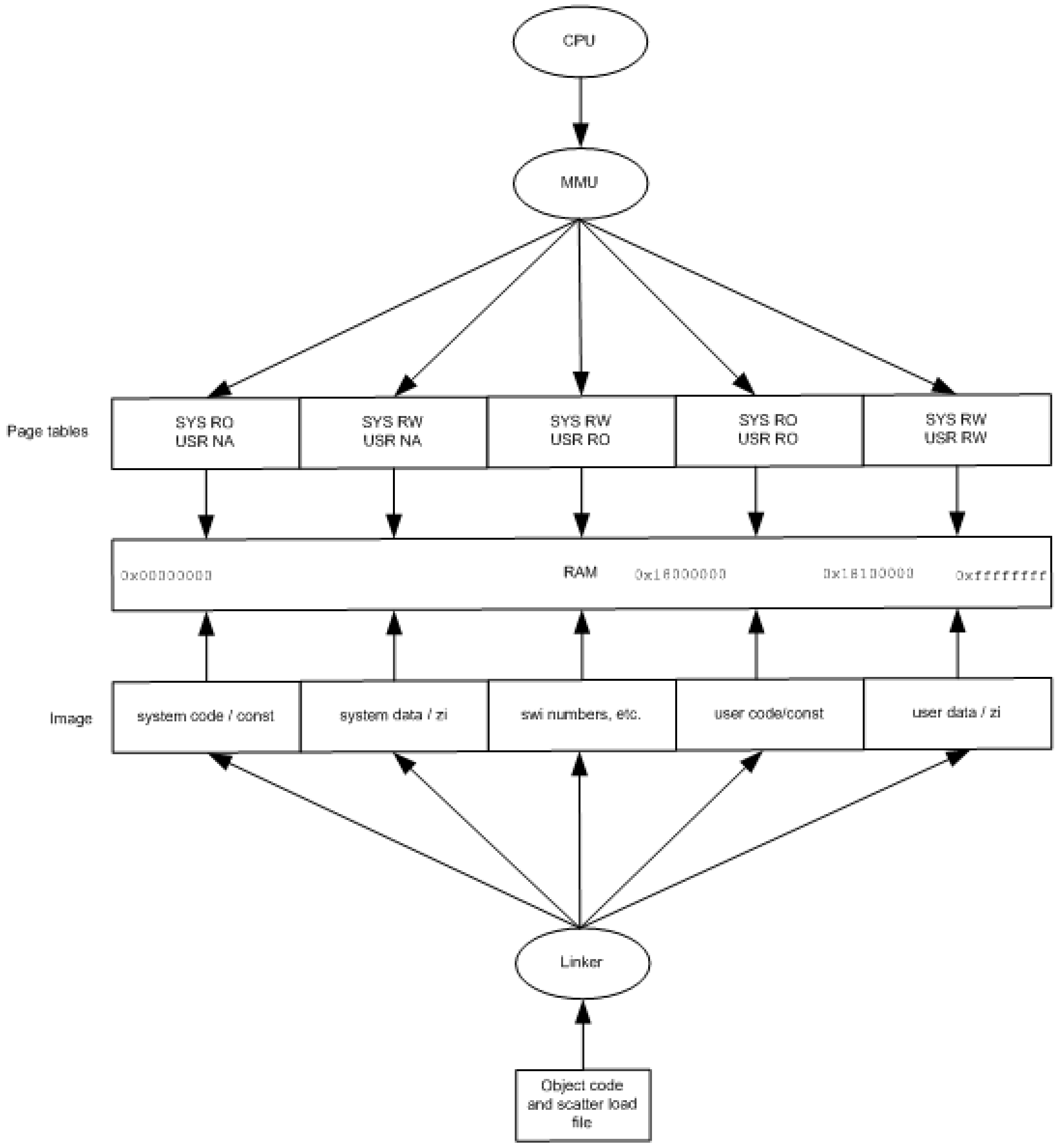
Once the MMU/MPU is configured, the global boolean flag *bridle_on* needs to be set to TRUE for BRIDLE to be enabled on the device. This allows the OEM to enable BRIDLE at runtime with no change in the BREW libraries.

## Scatter load

The ARM scatter load file is a collection of rules that describe which portions of object code result in which regions of the address space. Since BRIDLE-I requires coherence between the code layout and the MMU/MPU configuration, the scatter load plays a very important role.

The following figure shows the relationship between the MMU and the scatter load.

### MMU and scatter load relationship



To support BRIDLE, you need to follow a small set of rules while creating your scatter load file; typically, this file has a .scl extension.

The total code and data memory must be divided into three types of regions as described in the previous section. There may be more than one region of the same type. The regions must be aligned to the nearest page or section boundary (MMU) or the nearest MPU region boundary. This process may waste some of RAM or flash. However, you can minimize the waste by optimizing the page or region type, and by reordering the code layout in the scatter.

The code and data sections from AEE libraries must be placed in appropriate regions. BREW has the following format for defining regions for its sections:

> bridle_svc_*type*_usr_*access,* where type describes the type of data that is present in that region and access describes the access permissions for that region when the processor is in User mode.

Possible values for type are shown in the following table.

| | |
|---|---|
| ro | Read-only |
| rw | Read/write |
| zi | Zerio-initialzed |

Possible values for access are shown in the following table.

| | |
|---|---|
| rw | Read/write |
| ro | Read-only |
| na | No access |

In addition to placing these BREW library sections in the appropriate regions in the scatter load, there are some more special BREW regions that must be placed as specified below:

> bridle_critical_section_zi: This data pertains to the various critical sections that may be used by BREW or OEM code to provide thread-safe access. Sections marked bridle_critical_section_zi must be in a Supervisor read/write, User no-access region.
>
> protected_svc_heapbytes: This is the BREW kernel heap. All calls to malloc when in SVC mode allocate memory from this heap. This section must be in a Supervisor read/write, user no-access region.
>
> unprotected_brew_heapbytes: This is the BREW heap. BREW applications as well as User mode services of BREW allocate memory from this heap. This section must be in a Supervisor read/write, user read/write region.

bridle_user_stack: This is the stack used by BREW in User mode. BREW applications as well as User mode services of BREW execute on this stack. This section must be in a Supervisor read/write, user read/write region.

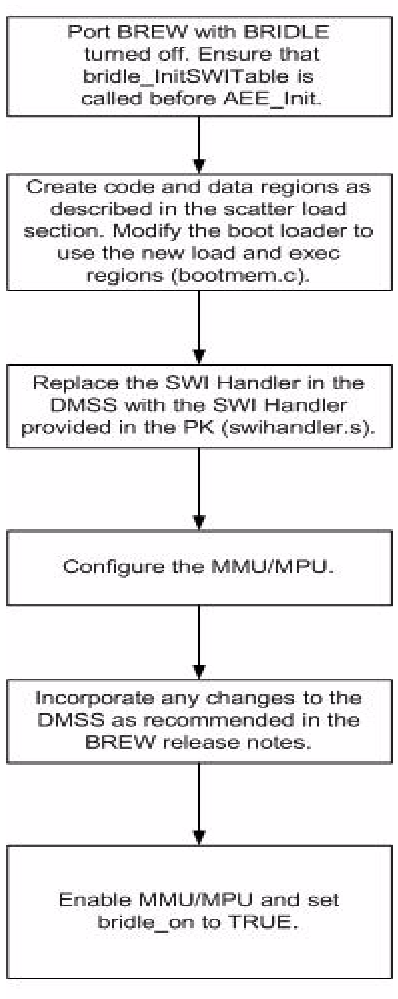
The following example illustrates a typical scatter layout for a device with an MMU or MPU, 4 megabytes of code space and 2 megabytes of RAM. The code space begins from offset 0, and the RAM starts at address 0x14000000. In this configuration, all code is read-only in User mode. The RAM is divided as follows: The first 16 kilobytes are read-only in User mode, and the remaining portion of the first 512 kilobytes is read-write in User mode. The upper 1.5 megabytes of RAM are not accessible in User mode. The file BREWMPU.c in the BREW PK illustrates how to configure the MPU (for ARM-7 based MSMs such as the 6050 and the 6200) for the above configuration:

```
CODE_ROM 0x0 0x3fffff
{
    SCATTER_U_RO_VBB_ROM 0x0
    {
      bootsys.o (BOOTSYS_IVT, +FIRST)
      bootsys.o (BOOTSYS_BOOT_CODE)
      bootsys.o (BOOTSYS_DATA)
      bootsys.o (BOOT_RAM_TEST)
      boothw_6050.o (+RO)
      bootmem.o (+RO)
      dloadarm.o (+RO)
      dloaduart.o (+RO)
      dloadusb.o (+RO)
      boot_trap.o (+RO)
      crc.o (+RO)
    }

    SCATTER_U_RO_MAIN_APP +0x0
    {
      * (+RO)
      * (bridle_svc_ro_usr_ro)
    }

   SCATTER_U_RO_BREW_USER_RO_RAM 0x14000000
   {
      * (bridle_svc_rw_usr_ro)
      * (bridle_svc_zi_usr_ro)
      bootapp.o (BOOTAPP_IVECT, +FIRST)
      bootbrew.o (BREWDATA_ABORT)
      *mif.o (+RO)
      *bar.o (+RO)
      *mod.o (+RO)
   }

   SCATTER_U_RW_BREW_RAM 0x14004000
   {
      * (bridle_svc_rw_usr_rw)
      * (bridle_svc_zi_usr_zi)
      * (unprotected_brew_heapbytes)
      bridle_stack.o (bridle_user_stack, +FIRST)
      sys_palette.o (+RW, +ZI)
      OEM*.o (+RW, +ZI)
      *AEE*.lib (+RW, +ZI)
      *BREW.lib (+RW, +ZI)
      *PNG.lib (+RW, +ZI)
      *z.lib (+RW, +ZI)
      *.l (+RW, +ZI)
      bridle_user.o (+RW, +ZI)
   }

   SCATTER_U_NA_APP_RAM 0x14080000
   {
      * (protected_svc_heapbytes)
      * (bridle_svc_rw_usr_na)
      * (bridle_svc_zi_usr_na)
      * (bridle_critical_section_zi)
      AEEBridle.lib (+RW, +ZI)
      OEMFS.o (+RW, +ZI)
      OEMSock.o (+RW, +ZI)
      OEMMutex.o (+RW, +ZI)
      OEMCriticalSection.o (+RW, +ZI)
      OEMPosDet.o (+RW, +ZI)
      OEMTAPI.o (+RW, +ZI)
      OEMSIO.o (+RW, +ZI)
      * (+RW, +ZI)
   }

   SCATTER_U_NA_BB_RAM +0x0
   {
      dloadarm.o (+ZI)
      dloadusb.o (+RW, +ZI)
      bootmem.o (+ZI)
   }

}
```

## Implementing BRIDLE integration

The following diagram shows the BRIDLE porting process

### BRIDLE Porting Process


Port BREW with BRIDLE turned off. Ensure that bridle_InitSWITable is called before AEE_Init.
Create code and data regions as described in the scatter load section. Modify the boot loader to use the new load and exec regions (bootmem.c).
Replace the SWI Handler in the DMSS with the SWI Handler provided in the PK (swihandler.s).
Configure the MMU/MPU.
Incorporate any changes to the DMSS as recommended in the BREW release notes.
Enable MMU/MPU and set bridle_on to TRUE.

## Verifying implementation of BRIDLE integration

Use OAT to verify your implementation of BRIDLE integration. OATBridle tests the MMU/MPU configuration and general ARM processor specific parameters, such as processor modes, interrupt manipulation, and illegal SWIs. See the *BREW™ Porting Evaluation Kit User Guide* in the PEK documentation set.

# BREW UI Guidelines

This section explains the BREW UI and discusses best practices for integrating your native UI applications in the BREW environment.

Many of the concepts and file types referred to in this section are covered in Managing and Downloading Applications and Extensions on page 77.

## Creating a static application or extension

This section describes how to create a static application or extension that can be exposed to BREW applications.

The first step in creating a static extension or application is to obtain a ClassID for the new extension or application.

### Obtaining a ClassID

A ClassID is a unique, 32-bit identifier for each BREW application or extension. It is provided through the BREW ClassID (BID) file (for example, MyExt.bid or MYApp.bid). The BID file also contains a name for the ClassID that was requested (for example, AEECLSID_MYAPP).

The recommended method for obtaining a ClassID is to request one from the OEM Extranet. If you are unable to obtain a ClassID from the OEM Extranet, follow the procedure described below.

#### To obtain a ClassID

- Assign a value from a set of ClassIDs for OEMs. In the OEMClassIDs.h file, there are two definitions for OEM ClassIDs (for example, AEECLSID_OEM and AEECLSID_OEM_APP). These two values define the start of a range of ClassIDs reserved for OEMs' extensions and applets.

- For a static extension, assign a ClassID that is in the range of (AEECLSID_OEM to AEECLSID_OEM_APP). Some ClassIDs in this range are already taken for the reference OEM code (for example, display). If you are using any OEM reference code that uses these OEM ClassIDs, you must offset your OEM ClassIDs to prevent a conflict from occurring.
- For a static application, assign a class ID that is in the range of (AEECLSID_OEM_APP to AEECLSID_SAMPLE_APP).

**NOTE:** Ensure that the new ClassID does not clash with an existing ClassID.

## Creating the extension or application

After you have obtained a ClassID, follow the procedure below to create the static application or extension.

### To create a static extension or application

1. To include the extension or application as a feature, introduce a new feature definition in OEMFeatures.h (for example, #define FEATURE_MYAPP).
2. Create a MIF for your application or extension using the MIF Editor provided as part of the BREW SDK™ (for example, MyApp.mif ).

   **NOTE:** The MIF filename must begin with an alphabetical character, not a number.

3. Create a persistent file entry for the MIF by executing the bin2src utility on the MIF to produce a source version of the MIF binary. For more information on persistent files, see Creating persistent files on page 107, and for information on the bin2src utility, see the *BREW PK Utilities Guide* . For example, if your MIF file is named MyApp.mif, then execute the following:

   ```
   bin2src -sMyApp.mif -dfs:/mif
   ```

   *This creates a file called MyAppmif.c.*

4. Include the MyAppmif.c file in your build.
5. Add the global variable declaration of MyAppmif.c to OEMConstFiles.c as follows:

   ```
   #ifdef FEATURE_MYAPP

   extern AEEConstFile      gMyApp_MIF;

   #endif
   ```

6. Add the address of gMYApp_MIF to the gpOEMConstFiles table as follows:

```
static const OEMFSPersistentFile * gpOEMConstFiles[] = {

...

#ifdef FEATURE_MYAPP

&gMyApp_Mif,

#endif

...

NULL};
```

7. Modify OEMModTableExt.c to associate your MIF to your code. This involves removing the MyApp_GetModInfo function from your static application (if it is already present) and exposing the MyApp_Load function that is normally static in that code. Then add an entry to the OEMStaticModstable.

   a. Add a declaration in OEMModTableExt.c for the Load function as follows:

   ```
   #ifdef FEATURE_MYAPP

   extern int MyApp_Load(IShell *ps, void * pHelpers, IModule **
   pMod);

   #endif
   ```

   b. Add the Load function to the gOEMStaticModList[] list as follows:

   ```
   const AEEStaticMod gOEMStaticModList[] =

   {

   ...


   #ifdef FEATURE_MYAPP

   {"MyApp.mif", MyApp_Load},

   #endif



   ...

   {NULL, NULL}

   };
   ```

8. Ensure that the following functions are defined for your application.

   a. In your application source file (for example, MyApp.c), declare the following:

   ```
   int  MyApp_Load(IShell *ps, void * pHelpers, IModule ** pMod);
   ```

```
   int MyApp_CreateInstance(AEECLSID ClsId,IShell * pIShell,IModule
   * po,void ** ppObj);
```

**b.** In the same file, define the MyApp_Load and MyApp_CreateInstance functions:

```
int MyApp_Load(IShell *ps, void * pHelpers, IModule ** pMod)
{
   return(AEEStaticMod_New((int16)(sizeof(AEEMod)),ps,pHelpers,
pMod,,MyApp_CreateInstance,NULL));
}
int MyApp_CreateInstance(AEECLSID ClsId,IShell * pIShell,IModule *
po,void ** ppObj)
{
   *ppObj = NULL;
   if(ClsId == AEECLSID_MYAPP)
   {
      if(AEEApplet_New(sizeof(MyApp), //Size of your private class
         ClsId,                       //Your Class ID
         pIShell,                     //Shell Interface
         po,                          //Module instance
         (IApplet**)ppObj,            //Return object
         (AEEHANDLER)MyApp_HandleEvent, //Your App's event handler
         (PFNFREEAPPDATA)MyApp_FreeAppData) //Cleanup function
      == TRUE)
     {
         // Invoke any Init function as needed
         return (AEE_SUCCESS);
     }
   }
   return (EFAILED);
}
```

*After you compile and link, your module is available in the build.*

**NOTE:** From this point on, your application is governed by the privileges set in your application's MIF.

## Including multiple applications or extensions

Each BREW module contains one or more BREW classes. These classes are either applet and non-applet classes (extensions) and are identified by ClassIDs. The applet classes are run by the device user, whereas the non-applet classes implement services used by the module's applet classes. These services are available to the classes of other modules as well. The MIF file contains unique ClassIDs for each of the module's classes and specifies which classes are exported for use by other modules.

For more information on creating extensions, refer to the Extending BREW APIs section in the *BREW SDK User Docs*, provided as part of the BREW SDK.

### To include multiple applications or extensions in one module

1. Obtain a ClassID for each application or extension that is part of the module being created (see To obtain a ClassID on page 103).
2. Using the same method, obtain a ClassID for each of the applets or extensions that are part of MyNewApp (for example, MyNewClass1.bid, MyNewApp2.bid).
3. The recommended method for including the module as a feature is to introduce a new feature definition in OEMFeatures.h (for example, #define FEATURE_MY_NEW_MOD).
4. Create a MIF for your module using the MIF Editor provided as part of the BREW SDK (for example, MyNewMod.mif ).

   **NOTE:** The MIF filename must begin with an alphabetical character, not a number.

   As part of creating the MIF, include the ClassID of the applets or extensions that are part of the module. For more information on creating MIFs, see the MIF Editor documentation provided as part of the BREW SDK User Docs.
5. Execute the set of steps starting from step 4 of the procedure To create a static extension or application on page 104.

## Creating persistent files

In BREW v3.0, the concept of persistent files is introduced. The term *persistent* indicates that the content remains in memory at all times and cannot be deleted unwillingly. A persistent file typically resides as static data in memory and not as a physical file on the file system tree. This provides a way to protect the file contents from being deleted or modified maliciously.

To create persistent files, a new tool named bin2src is provided (see the *BREW PK Utilites Guide*). This tool takes as input a binary file and creates as output a source file which is compiled into the handset software image. An entry for this file can be added into the OEMConstFiles.c file. The OEMConstFiles.c file contains the data structures populated by the OEM with the list of persistent files. These files consist of data structures that are accessed by OEMFS.c as virtual files. This allows the OEM to reduce RAM overhead by placing persistent files (AEEControls.bar, and the like) into virtual const data that consumes code space but not RAM.

## Registering a handler

Use ISHELL_RegisterHandler() to associate a MIME type with the ClassID of the BREW handler class you implemented to handle this type. To update a handler, delete the existing handler from the database by calling ISHELL_RegisterHandler() with a 0 (zero) ClassID. See the *BREW API Reference Online Help* for more details on this API.

The following shows how ISHELL_RegisterHandler() deregisters and registers a MIDI MIME type:

```
// De-register any existing handler for this MIDI MIME type
// MT_AUDIO_MIDI = "audio/mid"
ISHELL_RegisterHandler(pIShell, AEECLSID_MEDIA, MT_AUDIO_MIDI, 0);

// Register this class as the handler for the MIDI MIME type
ISHELL_RegisterHandler(pIShell, AEECLSID_MEDIA, MT_AUDIO_MIDI,
AEECLSID_MEDIAMIDI);
```

ISHELL_GetHandler() returns the ClassID of the handler class associated with a given MIME type. See the *BREW API Reference Online Help* for more details on this API.

The following shows how ISHELL_GetHandler() gets the classID for the MIDI MIME type that was registered earlier.

```
// The return value in this case would be AEECLSID_MEDIAMIDI
clsID = ISHELL_GetHandler(ps, AEECLSID_MEDIA, MT_AUDIO_MIDI)
```

# Using and extending INotifier

Notification is a mechanism that notifies classes when certain events occur in other classes. The listener registers for notifications by using ISHELL_RegisterNotify(), so it receives an EVT_NOTIFY event when the event occurs.

Notifiers are classes that generate and dispatch notifications when certain events occur. Whenever a notifier sends notifications, it uses ISHELL_Notify(). The INotifier interface provides the functions to any class that sends notifications to other applets. See the *BREW API Reference Online Help* for more details on this interface. Functions in the INotifier interface include:

- INOTIFIER_AddRef()
- INOTIFIER_Release()
- INOTIFIER_SetMask()

You define the set of notifications, or masks, that the notifier class can issue. These masks must be available to other applets so the applets interested in these notifications can register for them. You can build the notifier class as either a static or dynamic module.

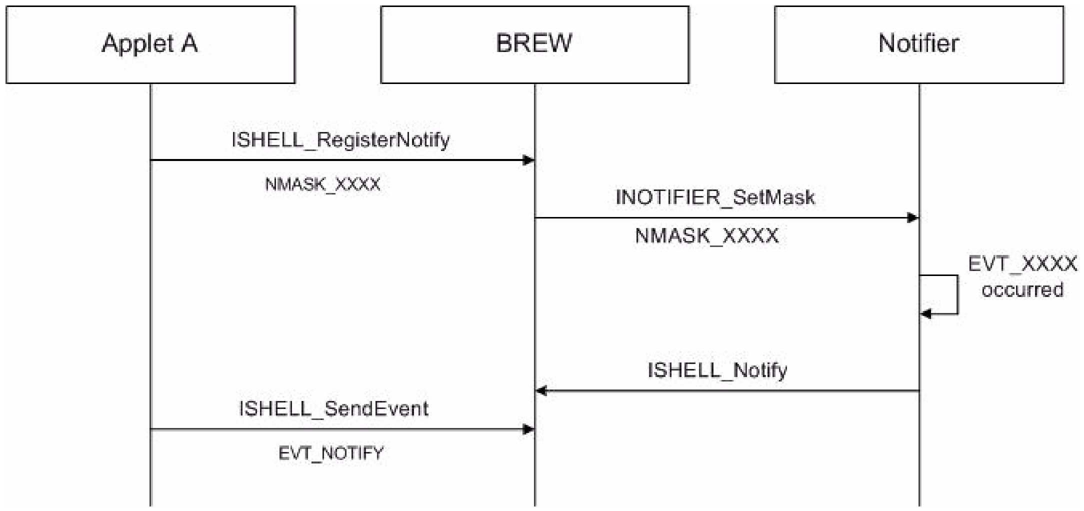
## Notification scenario

Consider a user, applet A, that must be notified by the shell when an event occurs on notification service B.

Applet A registers with the shell by invoking the ISHELL_RegisterNotify() function with the event corresponding to NMASK_XXXX on notification service B. (The applet registers for this event in the MIF also.) The shell sends a request to the notification service B to inform the shell when an event corresponding to NOTIFY_MASK_XXXX occurs on service B. Service B then adds this event to its list of notify events.

When an event on the service B occurs, it reviews the notify event list and informs the shell that EVT_XXXX corresponding to NMASK_XXXX occurred by invoking the ISHELL_Notify() function. The shell reviews its registered entry list and sends an EVT_NOTIFY event to the appropriate registered applets. The dwParam of this event is of type AEENotify. The pData member in the AEENotify Structure contains notifier-specific data. See the call flow diagram below.

## Call flow



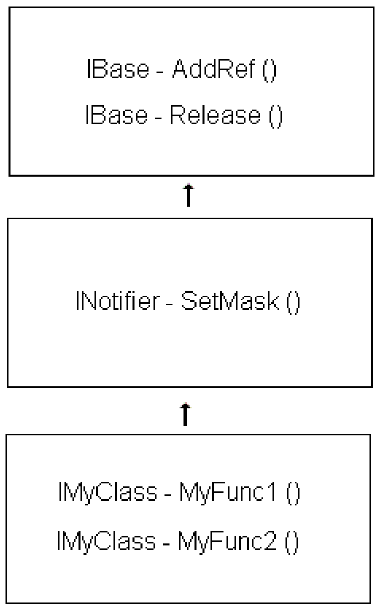
## Implementing an INotifier class

When you create a BREW class that will notify registered classes of certain events, you must call the macro INHERIT_INotifier. Example code follows, in which the new class is named IMyClass:

```
typedef struct _IMyClass IMyClass;

QINTERFACE(IMyClass)

{

   INHERIT_INOTIFIER(IMyClass);

   void (*MyFunc1)(IMyClass * po);

   int (*MyFunc2)(IMyClass * po, MyParm myParm);
}
```

Three methods are inherited from INotifier. Two of these methods are from IBase, as shown in the following diagram.

### *Methods for Implementing an INotifier class*



You must implement all three functions from the derived INotifier, in addition to the methods defined inside the new class (MyFunc1 and MyFunc2). For a detailed description of each function, see the *BREW API Reference Online Help.*

Refer to OEMTAPI.c for an example of this notification mechanism. OEMTAPI.c notifies registered applications of TAPI status changes.

## Extending IControl and creating an image viewer

IControl is the base class object from which all UI controls are derived. The interface provides common methods exposed by all UI controls. Because the interface is abstract, it is not possible to create an instance of the IControl interface directly. Create your own controls by inheriting from the IControl class and then adding control specific functions.

The following example shows how to extend the IControl interface to create an image viewer control interface (IImageCtl). This allows the caller to display a scrollable view in an image.

```
typedef struct _IImageCtl    IImageCtl;

QINTERFACE(IImageCtl)
```

```
{
    INHERIT_IControl(IImageCtl)

     void (*SetImage)(IImageCtl * po, IImage * pi);
     void (*SetRedraw)(IImageCtl * po, PFNNOTIFY pfn, void * pUser);
}
```

This means that the IImageCtl interface inherits all the controls from the base class IControl, implements all of the functions in the IControl Interface, and exposes the following functions:

- IIMAGECTL_SetImage: Sets image into the image control.
- IIMAGECTL_SetRedraw: Sets the redraw call back for the image set in image control.

For more details on IControl interface functions, see the *BREW API Reference Online Help*. You can create BREW custom controls to match your native controls, so both BREW and the native environments share the look and feel of a common set of controls.

### Implementing the custom controls

Refer to AEEImageCtl.c that provides the implementation for the IImageCtl interface by inheriting from the IControl class.

## Extending text control

BREW provides a standard way for applets to input language characters, numbers, and symbols. Because there is a diverse mix of devices that use many different languages, character sets, and mechanisms for inputting them, you must provide the underlying layer that takes care of character input and behavior for your specific devices. The ITextCtl interface is found in OEMitextctl.c.

### Implementing the text control interface

To implement the text control interface (ITextCtl), see source code files OEMitextctl.c, OEMText.c, and OEMText.h.

## Reference implementation

See the text control interface reference implementation in OEMitextctl.c, OEMText.c, and OEMText.h.

### Customizing reference implementation

For devices using English only without a third party text translation library, the text control mechanism doesn't need to be updated because the reference implementation delivered to you already performs all the functions required by BREW. For a non-English device or integrating a third party text translation library, you must modify the reference implementation to accommodate its own language character sets and the required behavior based on key inputs. The reference implementation provides the following three text input modes.

| Text input mode | Description | Contained in |
|---|---|---|
| Symbols | Input symbols cannot be displayed by other keystrokes. | OEMitextctl.c |
| Numbers | Input numeric characters from 0 to 9. | OEMText.c |
| Multitap | Input English alpha characters. | OEMText.ct |

#### To add a new mode (for example CUSTOM) for a new set of characters

**1.** Declare the mode in OEMText.c as follows:

```
#define TEXT_MODE_SYMBOLS AEE_TEXT_MODE_SYMBOLS
#define TEXT_MODE_NUMBERS (AEE_TEXT_MODE_USER+0)
#define TEXT_MODE_MULTITAP (AEE_TEXT_MODE_USER+1)
#define TEXT_MODE_CUSTOM   (AEE_TEXT_MODE_USER+2)
```

**NOTE:** (AEE_TEXT_MODE_USER+2) must be a unique number.

**2.** Change the structure **from** the following:

```
static const ModeInfo sTextModes[] =
{
{TextCtl_MultitapRestart,
    TextCtl_MultitapKey,
    TextCtl_MultitapString,
    TextCtl_MultitapExit,
    {TEXT_MODE_MULTITAP, {'M','u','l','t','i','t','a','p',0}}},
                        {TextCtl_NumbersRestart,
    TextCtl_NumbersKey,
    NULL, /* Use default name for Numbers mode */
    TextCtl_NumbersExit,
    {TEXT_MODE_NUMBERS, {'N','u','m','b','e','r','s',0}}},
   {TextCtl_SymbolsRestart,
    TextCtl_SymbolsKey,
    NULL, /* Use default name for Symbols mode */
    TextCtl_SymbolsExit,
    {TEXT_MODE_SYMBOLS, {'S','y','m','b','o','l','s',0}}}
};
```

**To** the following:

```
static const ModeInfo sTextModes[] =
{
{ TextCtl_MultitapRestart,
TextCtl_MultitapKey,
TextCtl_MultitapString,
TextCtl_MultitapExit,
{AEE_TM_LETTERS, {'M','u','l','t','i','t','a','p',0}}},
{ TextCtl_NumbersRestart,
TextCtl_NumbersKey,
NULL, /* Use default name for Numbers mode */
TextCtl_NumbersExit,
{AEE_TM_NUMBERS, {'N','u','m','b','e','r','s',0}}},
{ TextCtl_SymbolsRestart,
TextCtl_SymbolsKey,
NULL, /* Use default name for Symbols mode */
TextCtl_SymbolsExit,
{AEE_TM_SYMBOLS, {'S','y','m','b','o','l','s',0}}},
{TextCtl_CustomRestart,
TextCtl_CustomKey,
TextCtl_CustomString,
TextCtl_CustomExit,
{TEXT_MODE_CUSTOM, {'C','u','s','t','o','m',0}}},
};
```

3. Include whatever variables are required for the TextCtlContext type to support this custom text mode.

An instance of this type is created in the OEM_TextCreate() function when the text control is initialized.

## Verifying implementation

Use OAT to verify your implementation of the text control interface. See the *BREW™ OEM Acceptance Test Guide* in the PEK documentation set.

## Working with third party language

If you are using the ZiCorp text entry prediction engine in your build, the following steps will help you integrate the reference source shipped with the Porting Kit. If you are using another vendor's product, customization will be needed to implement the IIMUI interface specified in OEMIMUI.h for that product.

### To intergrate the reference source shipped with the Porting Kit if you are using ZiCorp

**1.** Add these files to your build:

   OEMIMM.h
   OEMIMUI.h
   OEMIMM.c
   OEMIMUI.c

**2.** Enable the features desired and supported by the eZiText library that are commented out in OEMFeatures.h.

   These features use OEMIMM.c and OEMIMUI.c:
   FEATURE_ZICORP_STROKE ZiCorp Simplified Chinese Stroke mode

   FEATURE_ZICORP_PINYIN ZiCorp Simplified Chinese Pinyin mode

   These features use OEMIMM.c:

   FEATURE_ZICORP_EZI_ENeZi English
   FEATURE_ZICORP_EZI_THeZi Thai
   FEATURE_ZICORP_EZI_ITeZi Italian
   FEATURE_ZICORP_EZI_FReZi French
   FEATURE_ZICORP_EZI_BPeZi Brasilian Portuguese
   FEATURE_ZICORP_EZI_PReZi Portuguese
   FEATURE_ZICORP_EZI_ESeZi Spanish

**3.** Specify any MobileShop less than or equal to 2.0.X.

4. Optionally customize the OEMText.c ←→IIMUI interactions and presentation.
5. Optionally customize the options used for obtaining characters in each mode of the IIMMgr.

   **NOTE:** If you do not use the Zi Corporation eZiText product, you must re-implement the IIMMgr to use the Key Translation Entry solution your handset provides.

## Modifications needed in the event handler for EVT_DIALOG_END

In Mobileshop.c Replace:

```
      case EVT_DIALOG_END:
         return MShop_SetState(pme, PREV_ENTRY);
```

With:

```
      case EVT_DIALOG_END:
     {
        if( pme->m_pText && (pme->m_wState == ST_SEARCH || pme->m_wState ==
ST_CREDITBACK) ){
            IDISPLAY_ClearScreen(pme->a.m_pIDisplay);
            if( pme->m_pSK ){
               IMENUCTL_Redraw(pme->m_pSK);
            }
            ITEXTCTL_Redraw(pme->m_pText);
            // Focus is in the TextCtl portion when Dialog is ended.
           MShop_SetTitle(pme, APPMGR_RES_FILE, (uint16)((pme->m_wState ==
ST_SEARCH) ? IDS_SEARCH_TITLE : IDS_CREDITBACK_TITLE), NULL);
            IDISPLAY_UpdateEx(pme->a.m_pIDisplay, FALSE);
         }
     }
     return TRUE;
```

This causes the screen to be redrawn after the additional UI presented for selecting a character is completed.

## Integrating native UI applications within BREW devices

The following process describes the set of steps you must perform to support the coexistence of BREW and non-BREW applications, also referred to as native applications, on the device. The primary goal is that the user should be able to switch from one application to the other regardless of whether the application is a BREW or native application. For example, the user switches from a game (BREW-based) to a browser (native application) and returns to the same instance of the same game when finished by using the browser.

The following scenario is an example of what can occur if you do not follow the instructions in the process.

1. Launch a BREW application (game).
2. Press a hot key to launch the browser (native application).
3. Press a hot key to launch the BREWAppMgr (a BREW application).
4. Using the AppMgr, try to resume the same instance of the game being played in step 1.
5. A new instance of the game is created, instead of resuming the previous instance.

One of the reasons for the problem described above is that you explicitly suspend and resume BREW by using AEE_Suspend() and AEE_Resume(). This section provides a method to avoid invoking these functions. Whenever the native software layer takes control of the display, a corresponding transient BREW application is started, so, for all BREW purposes, a BREW application is running. BREW maintains an application list, a list of all currently running applications and the order in which to resume them. Typically, the OEM layer maintains a similar application list for the native applications. It causes issues when you switch between native and BREW applications.

The following guidelines can be used in scenarios such as handling incoming calls when a BREW application is running, and handling incoming SMS when a BREW application is running. The goal of this procedure is to ensure that the functions AEE_Suspend() and AEE_Resume() are never invoked. Instead, the same must be accomplished by starting BREW applications. For example, if a BREW application is running and an incoming call comes into the system, instead of calling AEE_Suspend() and then putting up a native UI dialog, you must start another BREW application that corresponds to the native UI dialog and display the UI dialog in the context of the BREW application.

Two scenarios follow, illustrating the differences between call handling in pre-3.0 releases and in 3.0.x releases.

### Pre-3.0 scenario

1. While a BREW application (for example, HelloWorld) was running, the device received an incoming call.
2. AEE_Suspend() was called from the native software layer.
3. A corresponding UI dialog was put up, advising that the user was in the call.
4. The call ended.
5. AEE_Resume() was called to resume the application (HelloWorld) that was running when the call was received.

### 3.0.x scenario

1. As a BREW application (for example, HelloWorld) is running, the device receives an incoming call.
2. A transient BREW application starts by calling ISHELL_StartApplet().

   **NOTE:** The transient application is also referred to as the shim application. The purpose of this example application is to ensure that drawing to the screen is done in the context of a BREW application.

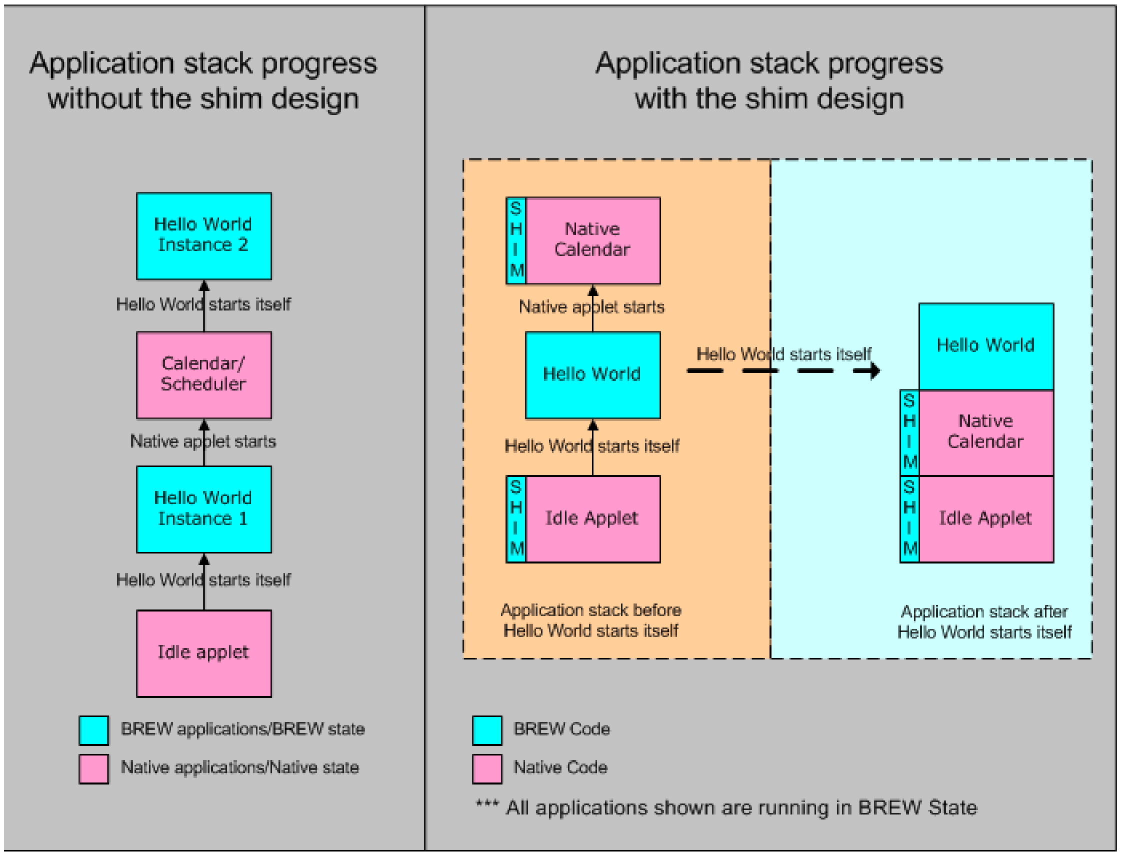
3. HelloWorld is automatically suspended.
4. The transient BREW application is used to put up a UI dialog corresponding to the call, advising that the user is in the call.
5. The call ends.
6. The transient BREW application is closed by calling ISHELL_CloseApplet().
7. BREW automatically resumes the application (HelloWorld) that was running when the call was received.

As shown in the example above, In the new procedures, you never have to invoke AEE_Suspend() or AEE_Resume() directly. This is automatically completed by BREW when other BREW applications are started. The goal is that there will always be a BREW application running on the device. This may be the transient BREW application you started to perform activities, such as display CallDialog, or it may be the BREW application that the end user invoked; for example, HelloWorld.

With the help of this section, only one application list is maintained on the device by BREW; this list is common to both BREW and native applications. This scenario helps BREW maintain and adjust the application list including the native applications, running each of the native applications in a BREW application context. This has a benefit of keeping a single application stack of BREW applications that can mix with native applications. These applications can all be moved in the application stack, except for the first application.

With this design, many complex UI interactions are possible between BREW and native applications as well as full BREW-based UI platforms that integrate applets not built on the BREW API. The following illustration shows a simple application stack comparison.

***.Simple application stack comparison***


Application stack progress without the shim design
Hello World Instance 2
Hello World starts itself
Calendar/ Scheduler
Native applet starts
Hello World Instance 1
Hello World starts itself
Idle applet
BREW applications/BREW state
Native applications/Native state
Application stack progress with the shim design
SHIM
Native Calendar
Native applet starts
Hello World
Hello World starts itself
SHIM
Idle Applet
Application stack before Hello World starts itself
Hello World starts itself
Hello World
SHIM
Native Calendar
SHIM
Idle Applet
Application stack after Hello World starts itself
BREW Code
Native Code
*** All applications shown are running in BREW State

There are similar considerations when an applet is meant to self-resume or start another applet already in a suspended state based on an event such as a SMS message or alarm. For example, if step 2 on page 118 consisted of an application already in the suspended state, receiving an EVT_APP_MESSAGE and requesting to start itself, it would resume, instead of having a new applet data created and maintaining a separate instance of the application.

To use the following procedure, you need a stable code base of a device with BREW completely ported. The affected files are OEMAppFuncs.c, OEMModTable.c, OEMHeap.c, and several native UI files. See the Reference code examples on page 79 and add these items to your device's build environment.

**NOTE:** The comments in the code labeled TODO: are the places in which you should consider customization.

### To integrate native UI applications in a BREW device

1. Integrate the objects from the code samples into your build.
   - Add staticapp.o to your build system.
   - Add oemidleapp.o to your build system.
   - Add oemtransientapp.o to your build system.
   - Arrange vpath and INC path to point to relevant areas in which these files were placed.
2. Modify the NUMBER_UI_APPLETS definition in shimapp.h to reflect the actual number of shim native applications, minus one for the idle applet that has its own applet structure.
3. Create a list of ClassID values for your native applications that run as a BREW shimmed application.

   The recommended practice is to put these ClassIDs in a header called shimapp.h. Add a ClassID reference to the first and last shimmed ClassID you use for an easier reference to see if a shimmed applet is currently running. These ClassIDs are used from the AEECLSID_OEM range defined in AEEClassIDs.h as follows:

```
#define AEECLSID_IDLEAPP (AEECLSID_OEM_APP) // Idle App
#define AEECLSID_CALCAPP (AEECLSID_OEM_APP+1) // Calc App
#define AEECLSID_MAINMENU (AEECLSID_OEM_APP+2) // Main Menu App 7
QUALCOMM Proprietary Problem Resolution Instructions
 // etc.
#define AEECLSID_FIRSTSHIM (AEECLSID_IDLEAPP) // First clsid
#define AEECLSID_LASTSHIM (AEECLSID_MAINMENU)// Last clsid
```

4. In oemtransientapp.c, add ClassIDs of the applications to be run with the BREW shim applet to the static AEEAppInfo structure except the idle applet's ClassID, since it has its own applet structure.

   This is shown with the above example in this code snippet:

```
static const AEEAppInfo gaiTransApp[NUMBER_UI_APPLETS] = {
{AEECLSID_CALCAPP,
NULL,0,0,0,0,0,AFLAG_POPUP|AFLAG_PHONE|AFLAG_HIDDEN},
{AEECLSID_MAINMENU,
NULL,0,0,0,0,0,AFLAG_POPUP|AFLAG_PHONE|AFLAG_HIDDEN},
};
```

**NOTE:** The reason these AFLAG_ properties are set is discussed later in the document.

5. Add the external Mod Info functions to the gGIList in OEMModTable.c as shown in this code snippet:

```
static PFNGETINFO gGIList[] = {

IdleMod_GetModInfo,
TransMod_GetModInfo,
NULL};
```

6. Define a function pointer that describes the function handler. A simple example is shown in the default shimapp.h as follows:

```
typedef ui_maj_type (*PFNUIEVENTHANDLER)(void);
```

**NOTE:** Your native event handler function may differ from this by taking an argument of the current event and extra data to pass with this, or another argument set altogether.

7. Create a map of the event handler function with the corresponding ClassID so your native event handler can look it up to invoke it as needed. Create this in oemtransientapp.c like the example below that uses the sample data above:

```
OBJECT(UIEventFnMap)
{
AEECLSID clsApp;
PFNUIEVENTHANDLER pfnEvtHandler;
};
static const UIEventFnMap gmapUIState[] =
{{AEECLSID_CALCAPP, uistate_calculator},
{AEECLSID_MAINMENU, uistate_mmenu}};
```

**NOTE:** uistate_calculator describes the native event handler function for the calculator main state or the calculator application. The same logic applies to uistate_mmenu. This map looks up the event handler function for this native application. It allows the native event handler to determine where to dispatch the event based on the current running applet.

8. To use the event handler first, include shimapp.h in your BREW state or substate. Call the function CShim_GetEventHandler() to retrieve the handler needed and the handler that is returned.
9. Add the idle application as the BREW auto-start application. This begins the BREW application stack after BREW is initialized with AEE_Init() and always remains as the base-level application. The idle application is persistent as the base application.
10. When all applications are closed, the auto-start application resumes. Include the shimapp.h header or the header in which you defined your ClassID list:

```
#include shimapp.h // Class ID of autostart
Handle the OEM_GetConfig() function for the case CFGI_AUTOSTART as
shown,
using the example ClassIds from above:
case CFGI_AUTOSTART:
{
AEECLSID * pc = (AEECLSID *)pBuff;
if( !pBuff || nSize != sizeof(AEECLSID) ){
return EBADPARM;
}
*pc = AEECLSID_IDLEAPP;
return AEE_SUCCESS;
```

Rather than transitioning into the idle state (IDLE_S), transition into your BREW state (BREW_S) or substate. Call your idle application's event handler directly after calling AEE_Init(), so the idle application starts its own initialization.

**NOTE:** If you have any other applets that must be started prior to the idle applet, the recommended process is not to run them under a BREW shim applet. This retains the uncomplicated logic of returning to the idle applet without adding several details to the idle shim for dispatching a launch to an idle applet when it is resumed. If some startup applets run as a UI state or as startup animation, idle initialization, address book initialization, or R-UIM initialization, allow them to run in a typical state machine manner and call AEE_Init() after the last initialization applet pops from the application stack.

You now have a sufficient framework to launch your idle application as a BREW shimmed applet. Before building, add a simple applet as a shim to see how a state pop and push transition works in the context of a BREW shimmed applet. Choose a simple applet that does not transition into many other states as an example to run as a shim. The example below involves a calculator application that has a state pop to close and a message state push to display a message. The message state can only pop. Choose a similarly simple application that doesn't branch off into several states or applications.

**NOTE:** For the next steps, assume that there is a UI state push and pop mechanism.

**11.** Ensure that the source file includes shimapp.h to implement the state push and pop mechanism. Make the edits that are suitable to your environment to achieve the effect shown below:

```
static void ui_state_push(ui_maj_type state)
{
if( uistack_pos >= UISTACK_SIZE-1 ){
ERR_FATAL( "Uistate stack full", 0, 0, 0 );
}else{
// If the state is the shimmed applet to be pushed, then push
// The BREW state, or leave the state alone if already BREW's state
// MY_SHIMMED_APP_S indicates the state value for the calculator
// state/applet in this example
if( state == MY_SHIMMED_APP_S ){
PFNUIEVENTHANDLER pfnEvtHndlr;
// UI_BREW_S reflects your actual BREW state or substate
if( uistack[uistack_pos] != UI_BREW_S ){
uistack[uistack_pos++] = UI_BREW_S; // Push a BREW state
// Depending on your environment you may need to call your
// BREW initialization routine here as well.
}
// Start the calculator application.
ISHELL_StartApplet(AEE_GetShell(), AEECLSID_CALCAPP);
// Look up the calculator application's event handler
if( (pfnEvtHndlr = CShim_GetEventHandler(AEECLSID_CALCAPP)) == NULL
){
// Error recovery code and return
}else{
pfnEvtHndlr();
}
}else{
uistack[ uistack_pos++ ] = state;
}
}
}
```

**NOTE:** You may need to make a map that correlates the ui_maj_type with the corresponding ClassID of the shim of that applet. This makes managing the transition easier than keeping a switch or if/else series.

This logic starts the shim applet when the shimmed state is pushed onto the stack, and the event handler is looked up, then invoked to start the applet.

**12.** Handle the pop functionality, which closes the applet if the state requesting to be popped is the BREW state and one of the shimmed applets is running.

**NOTE:** The BREW state can and must pop at some time, such as when the OEM_Notify() is invoked with OEMNTF_IDLE.

```
static ui_maj_type ui_state_pop( void )
{
if( uistack_pos == 0 ){
return( UI_NOSTATE_S );
}else{
// UI_BREW_S is a reference to the actual BREW state value you have
if( uistack[uistack_pos] == UI_BREW_S ){
AEECLSID clsApp = ISHELL_ActiveApplet(AEE_GetShell());
// AEECLSID_FIRSTSHIM and AEECLSID_LASTSHIM are references
// to the values added earlier when creating class ID for your
// shim applications.
if( clsApp >= AEECLSID_FIRSTSHIM && clsApp <= AEECLSID_LASTSHIM ){
// close this applet without returning to IDLE
// return no state as it will be ignored by the callee
// (shimmed BREW app) anyway
ISHELL_CloseApplet(AEE_GetShell(), FALSE);
return UI_NOSTATE_S; 13 QUALCOMM Proprietary Problem Resolution
Instructions
}
}
return( uistack[ --uistack_pos ] );
}
}
```

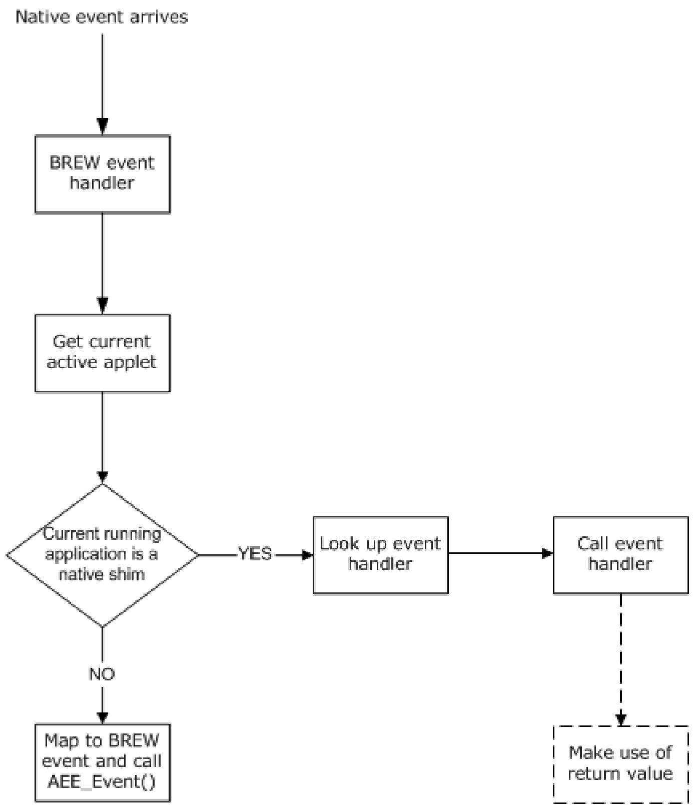
This process verifies if the current state is the BREW state requesting to be popped. If it is, a further check is made to see if the pop was triggered by a shimmed application. This method ensures that the BREW state can be popped when needed and closes the shimmed applet when needed as well.

**NOTE:** The steps above ensure that the state management mixes well between BREW applications, shimmed BREW applications, and native applications that are not shimmed. The next steps ensure that the shimmed applets receive all the events properly, as they may not have an exact correspondence with the BREW events.

**13.** When an event occurs, look up the current event handler based on the ClassID of the shimmed application that is running and invoke it with the arguments as needed.

The following flowchart shows the call flow of this step.

***Call flow.***


Native event arrives
BREW event handler
Get current active applet
Current running application is a native shim
YES
Look up event handler
Call event handler
NO
Map to BREW event and call AEE_Event()
Make use of return value

When an event comes in to the BREW state and the currently executing application is a native shimmed application or idle application, perform the lookup and call the event handler. Continue to send the BREW-related events to BREW as you usually do, and they'll be safely ignored by the shim. Ensure your event handler source file contains shimapp.h inclusion to access the event handler lookup function. See the code snippet below for an example of how these events are sent in case of the simple event handler:

```
static ui_maj_type BREWEventHandler()
{
AEECLSID clsActive = ISHELL_ActiveApplet(AEE_GetShell()); 14
QUALCOMM Proprietary Problem Resolution Instructions

// Near the end of the function send the native event
if( clsActive >= AEECLSID_FIRSTSHIM && clsActive <=
AEECLSID_LASTSHIM ){
PFNUIEVENTHANDLER pfnEvtHndlr;
// Look up the handler based on the active class ID
if( (pfnEvtHndlr = CShim_GetEventHandler(clsActive)) == NULL ){
// error recovery and return
} else {
pfnEvtHndlr();
}
}
 }
```

## Characteristics of shim applets

If you have to pay attention to the return value of the event handler, do so in the instances that invoke the event handler. Take the same action you usually take based on this return value, for example, launching a new application or closing the current application.

The following is an explanation of some of the properties and functionalities of these shim applets:

- AFLAG_POPUP is used so the screen does not clear when an applet starts. Often there are applets that do not occupy the full screen or perform some dither or fade on the screen in the background rather than erasing it fully.
- AFLAG_PHONE is used so the end key is processed by the shim applet and does not end the applet prematurely.

- AFLAG_HIDDEN is used to prevent the applet from being shown on the Application Manager or MobileShop®. Any shimmed applet you want present on the Application Manager may remove this flag.
- EVT_APP_RESUME may need to be inspected to see if the resume should trigger a call to the native event handler.
- EVT_KEY is handled with the clear key to prevent the applet from prematurely ending when the AVK_CLR key acts as the close application key in BREW. By treating it as handled, the shimmed applet processes it fully and, if needed, pops its state from the state machine. If your key to close an applet is different, modify the AVK_CLR to match the key defined to close an applet. See the *BREW OEM API Reference Online Help* for information about the application close keys.
- The static application helper files provide services to create applet contexts in a static method with almost no memory allocation, so shimmed applets run in a no memory situation as they are using static global space for the applet structure.

If your applet needs a situation where the current applet is closed before the next is started, this is achieved by using the EVT_CLOSE_SELF event in the following manner:

```
// Now we want to close current app before transitioning to
// the next applet. See the pseudo code below to achieve this.
ISHELL_PostEvent(pme->a.m_pIShell, pme->clsMe, EVT_CLOSE_SELF, 0,
0);
ISHELL_StartApplet(pme->a.m_pIShell, CLSID_OF_APP_TO_START);
```

**NOTE:** The next steps convert each application into a shim after you practice with the more simple applets previously described in this document. The routine is the same—building on the uistate.c, oemtransientapp.c and shimapp.h files for each new application you introduce above the shim layer.

### Low memory situations

Although some applications will close in the case of a low memory situation, their order on the application stack is maintained by BREW and they will be relaunched as needed by the application stack's determined ordering. Ensure that your idle application and other native applications do not try to persist function pointers and other free-able resources when they are closed. They must run safely when their resources are released. Also, it may be prudent to save off state information, so when the applets are relaunched the user will see the same state they left the application in when they return to it. To test low memory situations, try a simple allocation of a large value, such as MALLOC( 0xFFFFFFFF );. An alternative to letting your application close is to run it as a background applet when the request to close occurs. You must take great care with this method because of the following two situations. One is freeing as much resource as possible to help the allocation requested succeed. The other is managing the applet stack properly to pop the application off of the background applet list and so that it becomes active again when necessary. After an applet goes into the background application list, it will not be resumed unless an explicit ISHELL_StartApplet() is called with its ClassID.

**NOTE:** Shimmed applets are movable on the stack and interact on the same application stack as static and dynamic BREW applications.

### To verify the effective integration of native applications

1. Ensure that native applets run properly under the shim. Ensure that screen updates and other services are properly engaged while running under the BREW state for the shim applets.
2. Ensure that non-shim applets are properly switched between the application stack. This helps identify areas in the shim state that are unexpected at first. Eventually there are no applets that are not shimmed, except the startup or initialization applets, if any.
3. Ensure that the shimmed idle application is properly activated when necessary. You may have an event that returns directly to the idle application. If so, use ISHELL_CloseApplet(pme->a.m_pIShell, TRUE) to achieve the same effect.

## Privacy check removal

API function, ITAPI_MakeVoiceCall() and IPOSDET_GetGPSInfo(), perform operations that affect a user's privacy. The privacy policy for using these functions is set by a carrier or network operator. BREW implementation of these invokes OEM_CheckPrivacy(), a placeholder that enforces or customizes the privacy policy.

### Enabling the privacy prompt

The sample implementation of a privacy prompt dialog in OEMPrivacy.c is off (inactive) by default, guarded by FEATURE_BREW_PRIVACY_PROMPT. If you need to enable and customize this implementation, define the macro FEATURE_BREW_PRIVACY_PROMPT in OEMFeatures.h. In previous BREW versions, the privacy prompt dialog was on by default, but in this BREW version, it is off by default.

## OEM_Check Privacy

OEM_Check Privacy works asynchronously. This allows the OEM to display a message/prompt to the user or contact the network. The callback is then called with the code (0 - success) and the request will either proceed or fail based on the policy decision. See OEM_Check Privacy in the *BREW OEM API Reference Online Help*.

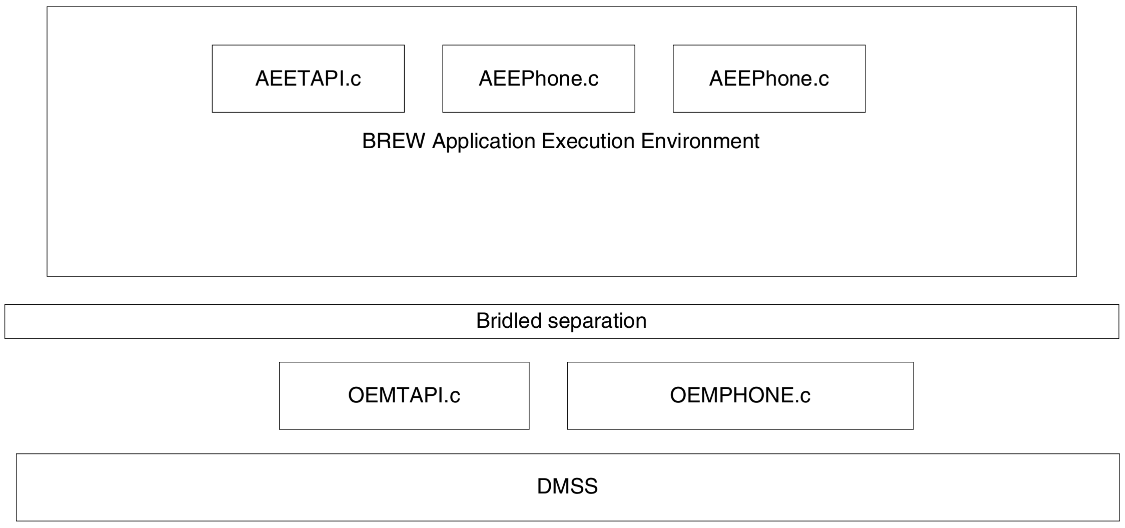
# Setting Up Call Handling

The ITAPI interface gives applications the ability to place voice calls, send short messages, and receive short messages. It also provides the services to query information regarding the device states. BREW implements the top layer of ITAPI interface. This implementation uses methods exported by OEMTAPI.h for the implementation of ITAPI telephony capabilities. ITAPI SMS capabilities are implemented by using ISMS, ISMSMsg and ISMSNotifier interfaces.

## Reference implementation

A reference implementation is provided in the following files: OEMTAPI.c, OEMPhone.h. The following illustration shows the layout of the implementation regarding applications.

***Layout of reference implemention regarding applications***



## Customizing reference implementation

The above OEM* and AEEPhone.c files are available in source.

## Verifying implementation

Use OAT to verify your implementation of the ITAPI interface. See the *BREW™ Porting Evaluation Kit User Guide* in the PEK documentation set.

## Call management

Typical device architecture contains a dialer module that presents the device states and incoming/outgoing call states.

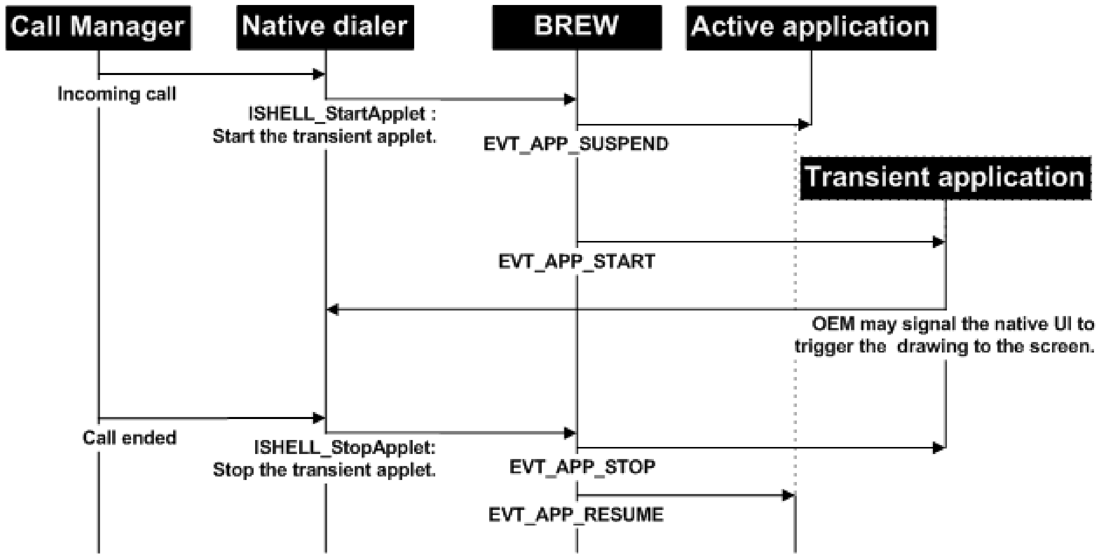
This section discusses the recommended mechanisms for handling device calls in conjunction with BREW. The recommendations are explained for the two different scenarios. In scenario (a), which shows a device with a non-BREW dialer, the dialer module is an external entity from BREW. See BREW-based UI or dialer on page 132. Scenario b shows a device with a UI dialer as a BREW application. See Managing call and position privacy on page 133.

### Handling incoming calls

This information applies to a device that does not have a BREW-based UI or does not integrate native applications within the BREW context.

If a call is received when a BREW applet is running, you must not suspend BREW. Instead, start a transient BREW application. Starting this application automatically allows you to suspend the current application, draw to the screen, and handle the keypad inputs. When the call is finished, the transient BREW application must be closed using ISHELL_CloseApplet()). This automatically resumes the suspended BREW application that was running when the call was received. See BREW-based UI or dialer on page 132.

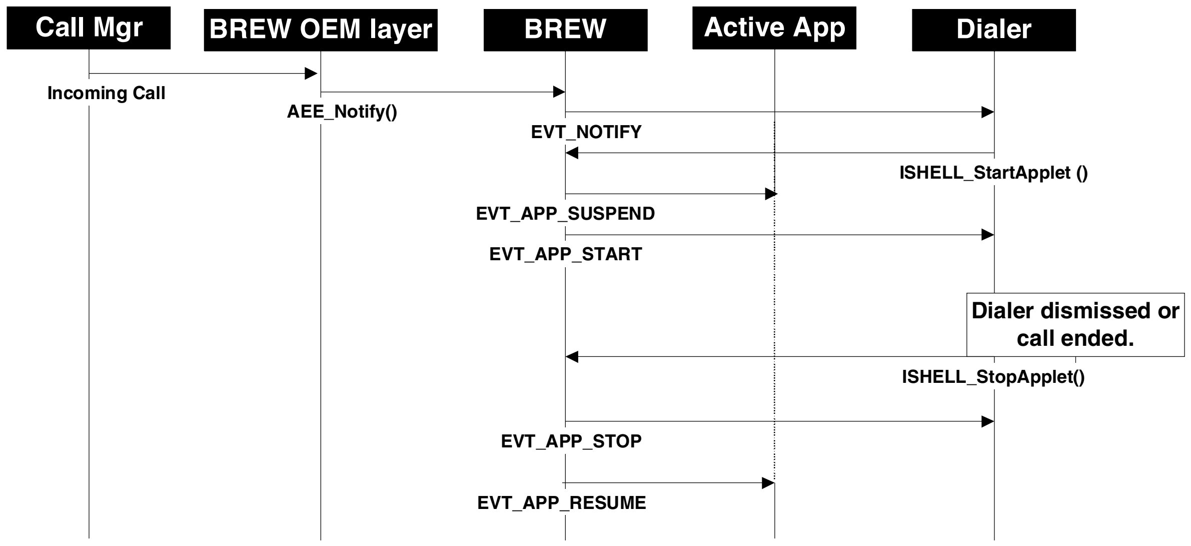
### *Call flow, scenario a: device with a non-BREW dialer*


Call Manager
Native dialer
BREW
Active application
Incoming call
ISHELL_StartApplet : Start the transient applet.
EVT_APP_SUSPEND
Transient application
EVT_APP_START
OEM may signal the native UI to trigger the drawing to the screen.
Call ended
ISHELL_StopApplet: Stop the transient applet.
EVT_APP_STOP
EVT_APP_RESUME

## BREW-based UI or dialer

The BREW-based dialer applet, instead of the transient applet, can be invoked directly to handle the call.

### *Call flow, scenario b, device with a BREW dialer*



## Handling outgoing calls from a BREW application

Handling outgoing calls from a BREW application is almost the same as handling incoming calls. The difference is that the privacy settings are checked before the call is placed. In the reference implementation, there is a code that displays a dialog informing the device user that a call is about to be placed. The device user can choose to allow or disallow the call. If the device user allows the call, the voice call is placed.

## Managing call and position privacy

You can block each originating voice call and position determination request based on application ClassID and privilege level by customizing OEM_CheckPrivacy(). Currently, BREW supports two privacy request types:

- Voice call (PRT_DIAL_VOICE): Passed in for voice call originations.
- Position determination (PRT_POSITION): Passed in for position determination requests.

For more information see Privacy check removal on page 128.

# Setting up SMS

The ISMS, ISMSMSg, ISMSNotifier, ISMSStorage, ISMSBCConfig and ISMSBCSrvOpts interfaces give applications the ability to send and receive SMS messages, store and retrieve SMS messages and configure user preferences for broadcast SMS. BREW implements the top layer of these interfaces. This implementation uses methods exported by OEMSMS.h and implemented by OEMSMS.c, OWMUASMS.c and OEMUASMSStorage.c on devices using uasms APIs for SMS and by OEMSMS.c and OEMWSMS.c on devices using wms APIs for SMS.

## Verifying implementation

Use OAT to verify your implementation of the SMS interfaces. See the *BREW Porting Evaluation Kit User Guide* in the PEK documentation set.

# Setting up battery

The IBattery interface gives applications the ability to query battery and charger-related information. The IBatteryNotifier interface enables applications to get notified about changes in battery and charger-related information. BREW implements the top layer of these interfaces. This implementation uses methods exported by OEMBattery.h and implemented by OEMBattery.

## Verifying implementation

Use OAT to verify your implementation of the IBattery interface. See the *BREW Porting Evaluation Kit User Guide* in the PEK documentation set.

# Managing Resources

## Managing Resources

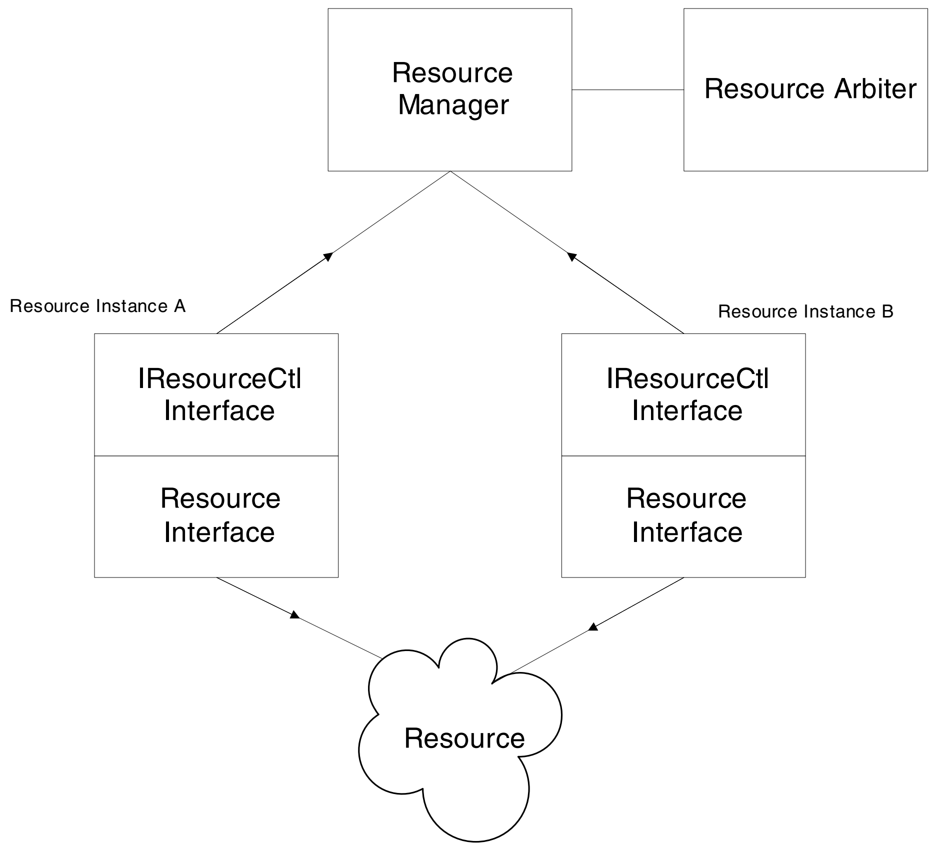
This section describes the BREW resource management mechanism. BREW currently implements top visible and ISound with resource management. Resource managment provides a generic means for objects, including BREW applications, to control resource access. The resource manager also coordinates and manages the acquisition and freeing of resources by objects and notifies registered objects when the state of a resource changes.

Some types of resources can only be used by one application at a time.  For example, only one application (the “top visible” application) writes to the display and receives the keypad events, or the ISound interface allows only one application to use the sound output. Resource management provides the arbitration (resource arbiter) for which an application is allowed to control a resource.

### Resource control architecture

For each resource being managed, there is a resource interface that controls the object, an IResourceCtl interface for controlling access, and a singleton resource manager.  The resource arbiter is shared among all resources.

## *Resource control architecture*

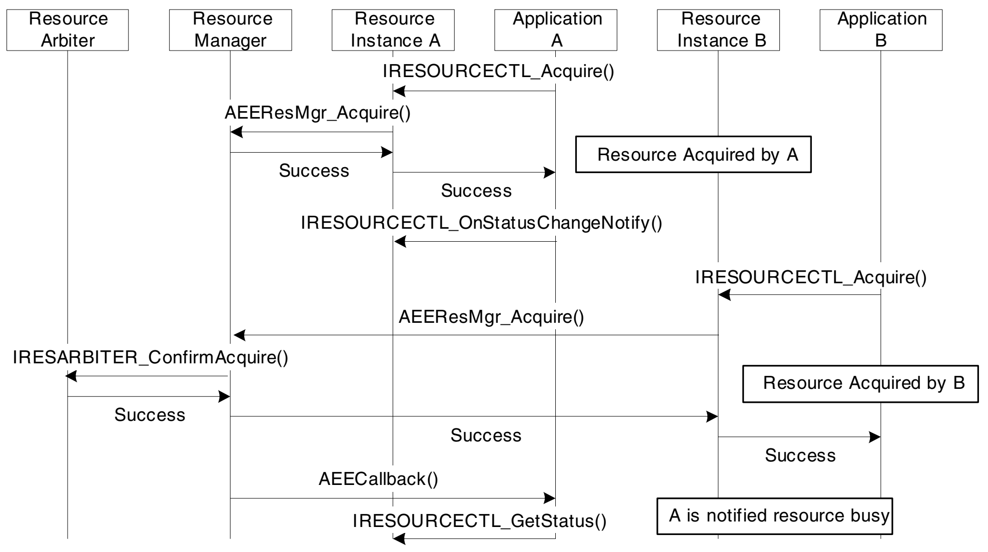


When you create an instance of the resource interface, it includes the IResourceCtl instance. The IResourceCtl instance interacts with the resource manager to acquire and free the underlying resource.

**NOTE:** Another application can take control of the underlying resource at any time.

In the following example, application A acquires the resource when it is not busy. Later, application B requests the resource, and it is granted. Since application A registered for status change notifications, it is alerted when the status changes.

### *Resource management example*



**NOTE:** To simplify the interface for most applications, so it doesn't need to explicitly acquire and free a resource, define the resource to automatically check and acquire the resource each time it uses it.

## Implementing a resource

For each resource being managed, for example for ISound, the following must be defined:

- an interface
- the associated AEECLSID for the interface
- an AEECLSID for the resource control so the resource manager can identify the resource
- an AEECLSID for the resource control group privilege. The group privilege ID is used to grant privileges to an application so that it can access the privileged features of the ResourceCtl, for example, being able to specify the relinquish list.

### Using IQUERYINTERFACE

When you create a BREW resource that supports resource management, it must be derived from the IQUERYINTERFACE using the INHERIT_IQUERYINTERFACE macro.  In the following sample code, the new class is named IMyResource.

```
typedef struct _IMyResource IMyResource;
QINTERFACE(IMyResource)
{
INHERIT_IQUERYINTERFACE (IMyResource);
...
}
```

As part of the inheriting process from IQUERYINTERFACE, the IQI_QueryInterface method must be implemented. This method must return the object of the type, IResourceC, associated with the resource when either AEECLSID_IRESOURCECTL or the app specific resource control ID is passed.

## Pointing to AEEResourceCtl

The second step in implementing resource management is including a pointer to AEEResourceCtl in your resource object. When creating the resource, use IResourceCtl_New() to initialize this pointer. Be sure to release this object when cleaning up.

## Implementing the IResourceCtl Interface

The next step is to implement the IResourceCtl interface. A default implementation for each of the functions is declared in the OEMResourceCtl.h header. If you wish to use these functions as is, declare your AEEVTBL with the following code:

```
static const AEEVTBL(IResourceCtl) gvtMyResourceCtl =
{
IResourceCtl_AddRef,
IResourceCtl_Release,
IResourceCtl_QueryInterface,
IResourceCtl_Request,
IResourceCtl_CanAcquire,
IResourceCtl_SetRelinquishCtl,
IResourceCtl_GetStatus,
IResourceCtl_OnStatusChangeNotify,
};
```

Otherwise you can override any or all of these methods with a custom implementation. For example, you could implement your own CanAcquire function to add additional privilege checks. The interface to AEEResMgr is provided in OEMResourceCtl.h for this purpose.

### Adding checks in the resource code

The final step in implementing resource management is to add checks in the resource code itself to make sure that the resource object is the current owner before accessing the underlying resource. The resource object is responsible for knowing when it can access the resource without conflict. For example, the ISound object checks that it is the owner before it sets the volume. Depending on your implementation needs, the ownership check could

- Check to see if it was the current owner and fail otherwise
- Check to see if it was the current owner, or if the resource is free, and fail otherwise
- Attempt to acquire ownership (you can acquire ownership even if you already own it) and fail if it can't acquire ownership.

## Customizing the resource arbiter

The resource arbiter is the central decision maker that determines if a resource can be handed over to the requesting object. The resource arbiter module is customizable by the OEM/Carrier and may be implemented as a downloadable module using the class ID, AEEIID_RESARBITER. There is a single IResArbiter implementation for all resources.

### Arbitration

The resource arbiter method, IResArbiter_ConfirmAcquire, is passed the following information to make its decision:

- Resource owner's information
  - Owner CLSID and instance pointer
  - Reason for acquisition
  - Relinquish control information
    - Relinquish ID list
    - List count (-1 == all, 0 == none, otherwise count)
- Requestor's information
  - Requestor CLSID and instance pointer
  - Reason for acquisition
  - Relinquish control information

- Relinquish ID list
- List count (-1 == all, 0 == none, otherwise count)

If the current owner has specified a relinquished CLSID list (see IResourceCtl in the OEM API Reference Online Help), and the requestor is in the list of IDs specified, or if the owner allows any ID (as in the case of a non-privileged owner), then the arbiter may decide to transfer ownership based on the rest of the information provided (the simplest implementation grants the request). If the requestor is not on the CLSID list, the arbiter rejects the request.  The following is a sample implementation of the ConfirmAcquire method for the resource arbiter (See OEMResArbiter.c).

```
int OEMResArbiter_ConfirmAcquire(IResArbiter * po, AEECLSID clsReq,
            AEEResCtlInfo * pOwner, AEEResCtlInfo * pRequestor)
{
      CResArbiter * pMe = (CResArbiter*)po;
      int status = EITEMBUSY;
      int i;
      //
      // first check class list to see if owner will allow it
      //
      switch (pOwner->nClsCount)
      {
case -1: // allow anyone to acquire resource
      status = SUCCESS;
      break;

case 0: // allow no one to acquire resource
      status = EITEMBUSY;
      break;

default: // check access (relinquish) list
for (i=0;i<pOwner->nClsCount;i++)
{
   uint32 privId = pOwner->pClsList[i];
   if (privId < QVERSION)
   {
   // is reason acceptable?
      if (privId == pRequestor->dwReason)
      {
      status = SUCCESS;
      break;
      }
}
else
{
      // does requestor class id match or has group privilege?
      if (ISHELL_CheckPrivLevel(pMe->m_pIShell,privId,TRUE))
      {
status = SUCCESS;
       break;
```

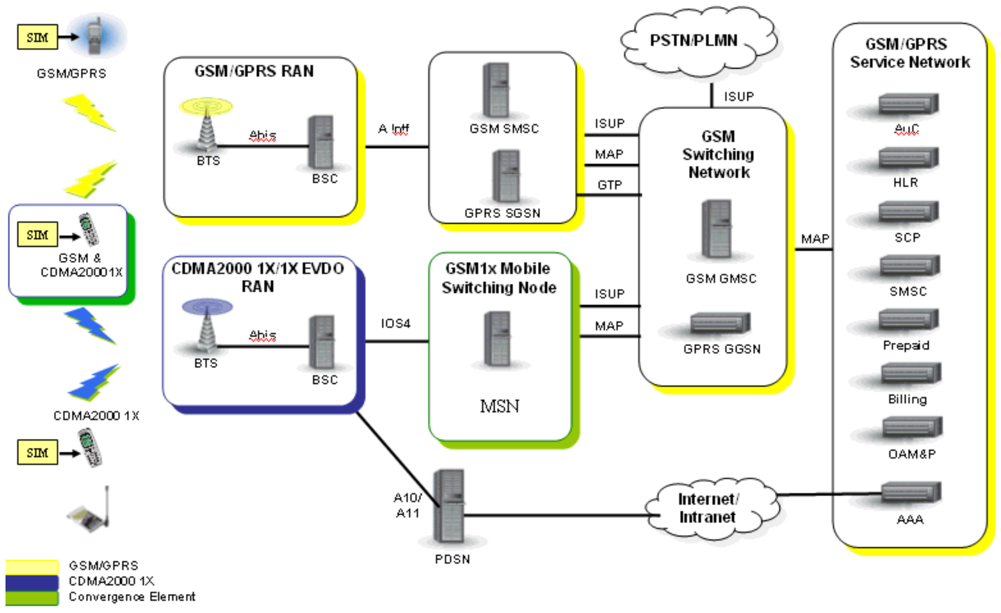
```
            }
         }
      }
      break;
   }
   // At this point, an OEM may choose to accept the access list permission
   // checks and/or add additional decision algorithms such as examining
   // current reason or allowing specific requestor clsid's reguardless
   // of the owner's access list, etc.

  // by default, if the current owner indicates it's busy (with dialogs or ?)
   // and the resource is TopVisible, don't release resource.
   // BREW sets the dwReason to RESCTL_REASON_BUSY if current application
   // responds to EVT_BUSY.
   if (pOwner->dwReason == RESCTL_REASON_BUSY && clsReq ==
AEECLSID_TOPVISIBLECTL)
      status = EITEMBUSY;

      return (status);
}
```

# Interoperability with GSM1x

## Introduction

QUALCOMM's GSM1x system architecture allows interoperability between a CDMA2000 radio access network and a GSM core network. A GSM1x device is a regular 1x device with a few software upgrades.

### *GSM1x system architecture*



## GSM1x requirements and recommendations

**NOTE:** The GSM1x mode is only possible if the device supports R-UIM or SIM cards. If the device has this capability, proceed with GSM1x.

QUALCOMM recommends that GSM1x support be enabled on a MSM6050 or above platform. If GSM1x support is needed for an older MSM platform, contact QUALCOMM.

The following DMSS features must be present:

- Sockets
- File system
- TAPI
- Operating system

## Prerequisites

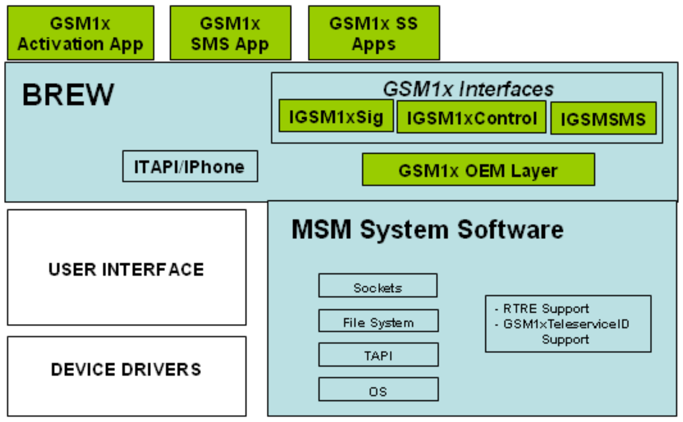
GSM1x developers need to familiarize themselves with the following.

- BREW software architecture
  - *BREW SDK™ User Docs* and *BREW™ API Reference Online Help*
  - GSM1x Engineering Specification Kit HB81-31494-2, Rev. B
  - *GSM1x Application User Guide*
    - Activation Application section
    - SMS Application section
    - Supplementary Services Application
  - *BREW Porting Evaluation Kit User's Guide*
  - *BREW OEM API Reference Online Help*

## Understanding GSM1x device architecture

The GSM1x software architecture is built on GMS1x-specific BREW interfaces which, when supported on the device, become GSM1x-capable. The GSM1x device software architecture figure below illustrates the software architecture of the GMS1x-enabled device.

### *GSM1x device software architecture*



The GMS1x BREW interfaces expose a common set of APIs at the BREW application layer. These APIs allow the GSM1x BREW applications to provide GSM1x capability on the handset. The GSM1x OEM layer in the figure above corresponds to modules that are device- or MSM platform-specific. The GSM1x OEM module ties the GSM1x BREW interface with the appropriate device or MSM software. For details regarding the GSM1x applications, see the *GSM1x Application User Guide.*

GSM1x device architecture depends on a feature called Run Time R-UIM Enable (RTRE). This feature must be enabled on the MSM software to support GSM1x mode on the device. The MSM software must support sending and receiving GSM1x Teleservice IDs (4104 - 4113).

**NOTE:** The above two features must be supported on the MSM software to enable GSM1x mode on the handset.

The examples section of the Porting Kit contains reference implementation for the GSM1x BREW applications. The GSM1x Application User Guide explains how the GSM1x applications operate.

## Understanding GSM1x BREW interfaces

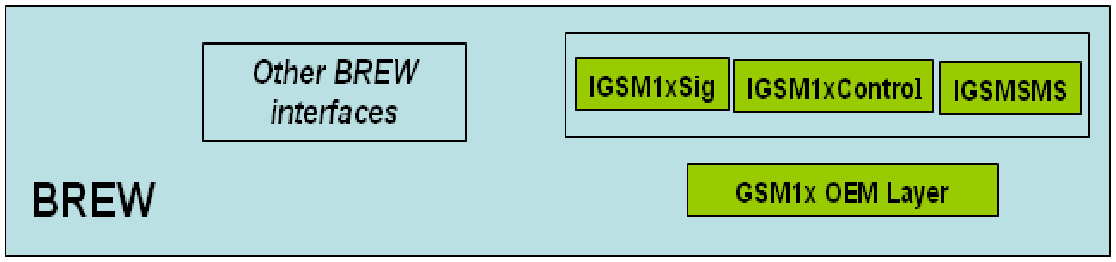
The GSM1x BREW interfaces perform the following general functions:

- Provide the capability to enable or disable the GSM1x mode on the device using the following interface:
  - IGSM1xControl BREW

- Exchange messages with the GSM core network to provide the GSM service layer transparency and authenticate itself using the following interfaces:
  - IGSM1xSig
  - IGSMSMS

The following figure illustrates a snapshot of the GSM1x interfaces.

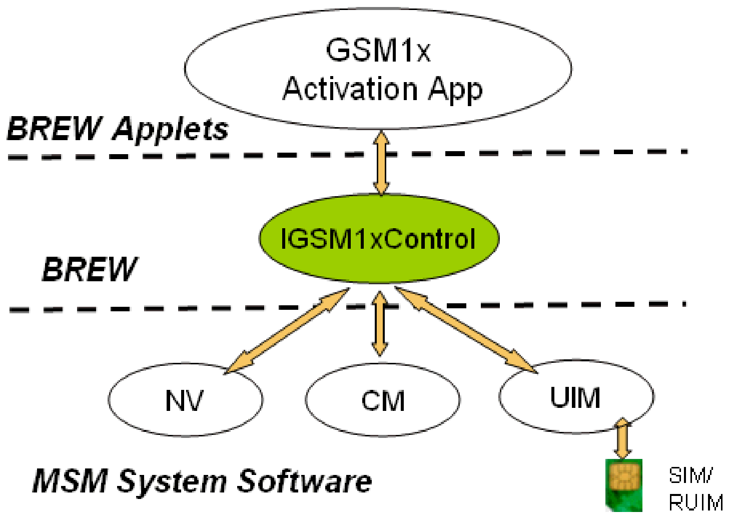
***GSM1x BREW interfaces***



## IGSM1xControl interface

The illustration below shows the IGSM1xControl architecture.

***IGSM1xControl architecture***



The IGSM1xControl interface provides the following functionality:

- Enables or disables the GSM1x mode.

- Performs GSM1x provisioning.
  - Reads the GSM user identity from DFgsm on the SIM/R-UIM.
  - Converts GSM identity parameters to CDMA1x identity parameters based on GSM1x algorithms.
  - Generates CDMA PRL based on PLMNs and available acquisition records.
  - Writes the obtained information to NV.
- Upon startup, determines if the GSM1x mode should be enabled. If so, it notifies GMS1x activation application.
- Signals all other GSM1x applications when the GSM1x mode is activated or deactivated.

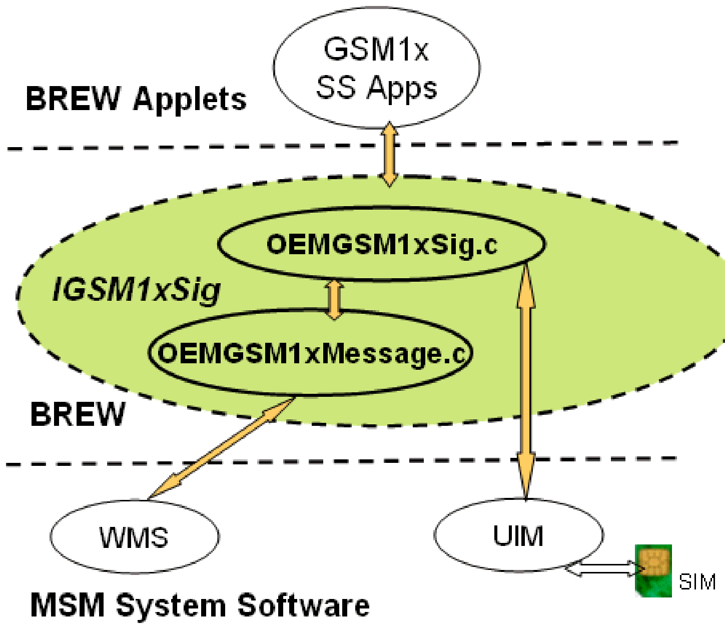
The following table shows source files and descriptions.

| **Source file** | **Description** |
|---|---|
| AEEGSM1xControl.h | IGSM1xControl interface specification |
| OEMGSM1xControl.c | Reference implementation for IGSM1xControl |
| OEMGSM1xProv.c | Implementation for GSM1x provisioning |
| OEMGSM1xCardHandler.c | Set of routines to read or write data to the GSM SIM |

## IGSM1xSig interface

The following illustration shows the IGSM1xSig architecture.

*IGSM1xSig architecture*



The IGSM1xSig interface provides the following functionality:

- Provide the ability to send and receive GMS1x signaling messages.
- Upon receiving an authentication request, implement GMS1x authentication by running GSM authentication on the GSM SIM and sending an authentication response.
- Use GSM1x Teleservice ID (4104) to send and receive GSM1x signaling information.

The following table shows source files and descriptions.

| Source file | Description |
|---|---|
| AEEGSM1xSig.h | IGSM1xSig interface specification |
| OEMGSM1xSig.c | IGSM1xSig reference implementation |
| OEMTAPI.c | Changes in OEMTAPI.c for GSM1x |

| Source file | Description |
|---|---|
| OEMGSM1xMessage.c | Routines to send and receive GSM1x messages |

### IGSMSMS interface

The IGSMSMS interface provides the following functionality:

- Provides a generic GSM SMS API.
- Supports reading and storing GMS SMS messages on the SIM.
- Uses the GSM1x Teleservice ID (4105) to send and receive GSM SMS messages.
- Provides BREW-directed SMS capability using the GSM SMS messages.

The following table shows source files and descriptions.

| Source file | Description |
|---|---|
| AEEGSMSMS.h | IGSMSMS interface definition |
| OEMGSMSMS.c | Reference implementation for IGSMSMS |
| OEMTAPI.c | ITAPI modification for GSM1x |
| OEMGSM1xCardHandler.c | Set of routines to read or write data to GSM SIM. |

## Implementing the GSM1x interfaces

GSM1x interfaces can be integrated with MSM software.

All current DMSS releases include a new feature, RTRE, which enables a device to dynamically select the provisioning source.

The following table shows how the device derives provisioning information based on the RTRE mode.

| RTRE mode | Provisioning source |
|---|---|
| NV_RTRE_Config_R-UIM_only | R-UIM |
| NV_RTRE_Config_NV_only | NV |

| | |
|---|---|
| NV_RTRE_Config_R-UIM_or_drop_back | Try R-UIM, if unsuccessful, use NV. |
| NV_RTRE_Config_SIM_access | SIM |

The IGSM1xControl interface relies on the RTRE feature to read provisioning information from multiple sources.

**NOTE:** To port IGSM1xControl to the device, RTRE in DMSS must be enabled.

The GSM1xControl interface uses the GSM1x Teleservice IDs (4104 - 4113)) to support porting efforts. It is enabled by defining FEATURE_GSM1x.

By supporting GSM1x mode, the mobile takes provisioning input from at least two sources, the GSM SIM for GSM1x and another for the CDMA2000 mode. For each provisioning source supported by the GSM1x mobile, a separate NAM must be assigned. Based on the selected provisioning mode, the appropriate NAM is used.

The following procedure provides the steps necessary to port the IGSM1x interface to a BREW-enabled device.

### To port the GSM1x interface

1. Ensure that the following features are enabled in the MSM software:
   - FEATURE_GSM1x
   - FEATURE_UIM
   - FEATURE_UIM_R-UIM
   - FEATURE_UIM_GSM
   - FEATURE_UIM_R-UIM_W_GSM_ACCESS
   - FEATURE_UIM_R-UIM_RUN_TIME_ENABLE
2. Ensure that feature flag FEATURE_GSM1x is enabled in OEMFeatures.h.
3. Allocate a signal in the task that BREW is running for use by OEMCardHandler.c. (This module has a set of routines to support read and write functions to GSM SIM.) Assign the signal value to OEMGSM1xPROV_UI_SIG_FOR_UIM in the OEMGSM1xProv.h file.

4. Allocate a NAM for holding GSM1x provisioning information. The GSM1x NAM can be the NAM used while the device is running with R-UIM or it can be a new NAM. Based on the value selected for GSM1x NAM, implement the function OEMGSM1xPROV_ReturnNAMUsedByProvisioningMode() in the OEMGSM1xProv.c file.
5. Implement the OEMGSM1xProv_IsModeSupported() function in the OEMGSM1xProv.c file, based on the provisioning modes supported by the device (for example, If the device supports both R-UIM and GSM1x modes, then return TRUE for both, OEMGSM1XPROV_GSM1X and OEMGSM1XPROV_GSM1x).
6. Implement or customize the reference implementation of SendRTRECommand() in the OEMGSM1xProc.c file.
7. Implement or customize the reference implementation of OEMGSM1xProv_SetESNUsage() in file OEMGSM1xProv.c.
8. Provide an implementation for OEMGSM1xProv_ActivateEmergencyCallOnlyState() in OEMGSM1xProv.c. This function is called if the device must go to emergency mode (for example, when the device can't locate a SIM or R-UIM, it goes to the emergency mode).
9. Call the function OEMGSM1x_Control_ProcessPowerUp() in OEMGSM1xControl.c from the task in which BREW is running. It should be called after the R-UIM or SIM card is accessible. (OEMGSM1xControl_ProcessPowerPowerUp() needs to read information from SIM or R-UIM, so this function should be called after completing PIN verification on the card.)
10. Enable support for the following data types in OEMConfig.h/OEMAppFuncs.c:
    - AEEGSM1xPRLInfo
    - AEEGSM1xRTREConfig
    - AEEGSM1xIdentityParams
    - AEEGSM1xSIDNIDParams
11. Customize the following GSM SMS-related parameters in OEMGSMSMS.c.

| **Parameter** | **Description** |
|---|---|
| GSMSMS_NV_ENTRY_SIZE | Size of GSM SMS entry stored in NV |
| GSMSMS_NUM_NV_ENTRIES | Number of GSM SMS stored in NV |

## Customizing reference implementation

The reference implementation for GSM1x interfaces is provided with the MSM Porting Kit and can be used as provided or customized as you need.

## Verifying Implementation

Use OAT to verify your implementation of the GSM1x interfaces. See the *BREW™ Porting Evaluation Kit User Guide* in the PEK documentation set.

# NTP Example BREW Extension

## Introduction

The Porting Kit Examples directory contains a sample implementation for a Network Time Protocol (NTP) BREW Extension. This extension is provided to support a method for devices to synchronize their local clock with a reliable network reference on wireless air interfaces that do not provide the time. The API is fully documented in AEENTP.h.

Handsets are required to have the local clock accurately set when attempting to connect to the BREW Application Download Server. BREW also uses the device's local clock for evaluating whether BREW time-based application licenses have expired. OEMs are responsible for deciding when to invoke INTP APIs for synchronizing the local clock.

## Integrating INTP as a Static Extension

OEMs need to perform the following steps

### To integrate the INTP extension into the handset build

1. Modify the makefiles to build NTP.c into the device build.
2. Add the following line to OEMModTableExt.c:

   ```
   extern int NTP_New(IShell *pShell, AEECLSID ClsId, void **ppObj);
   ```

3. Add the following line to the declaration of gOEMStaticClassList in OEMModTableExt.c:

   ```
   {AEECLSID_NTP, ASCF_UPGRADE|ASCF_PRIV, 0, NULL, NTP_New},
   ```

4. Determine where to invoke INTP APIs for synchronizing the local device clock.

# OEM Acceptance Process

This section describes a general process for integration and acceptance testing.

The device acceptance process varies from operator to operator, and typically includes many non-BREW test phases such as network performance, interoperability testing, and UI testing. Testing of the BREW port is often an additional phase, and requires the use of the BREW PEK.

For a description of the PEK and instructions for using it, see the *BREW Porting Evaluation Kit User's Guide*.

To help you achieve device acceptance on the BREW port, follow the steps below.

## To achieve device acceptance

1. Work with the operator to complete the BREW Device Requirements questionnaire.

   *This provides you with clear functional and performance requirements.*

2. Complete a Device Data Form/Device Pack using the Device Configurator.

   *This sheet is very valuable to the application developers who are producing BREW applications in support of the operator's launch of your device.*

3. Run and pass the tests in the PEK.

   *This extremely important step provides critical feedback about your completed porting process. PEK performs functional testing, measures performance benchmarking, and verifies conformance with the Device Data Sheet.*

4. Test the end-user experience.

   *QUALCOMM recommends that you run sample applications on the device and test the UI interaction between BREW and the native menus; for example, the management of ringers, address book contacts, wallpaper, screen savers, and so forth.*

# Static Modules

In previous BREW Porting Kit releases, OEM support for static modules and applications took the form of an entrypoint in the OEMModTableExt.c that referenced a function that returned information regarding static applets. It also returned a PFNLOAD function pointer used to create the associated IModule. This process created several problems:

- Static modules had limited information they could expose, as opposed to applications with MIFs.
- These applications and modules had no restriction on file system access and exhibited different file system behavior than dynamically downloaded versions.
- These applications always ran with system-level privileges.

BREW 3.0 enforces a new mechanism that mandates the use of MIF files for static applications and modules. Rather than special case handling of static modules, BREW enumerates all MIFs at startup. After all MIFs have been evaluated, the static module table is retrieved to determine if any of the MIFs found during initialization are associated with static code. If so, the static code is called for the module rather than attempting to load a dynamic module for the MIF.

## Associating MIF files with static modules

To associate MIF files with static module code, add an entry to a table in the OEMModTableExt.c file. An example of this table follows:

```
-- OEMModTableExt.c -----------------------------

extern int StaticApp_Load(IShell *ps, void * pHelpers, IModule ** pMod);

static const AEEStaticMod gOEMStaticModList[] =
{
   {"fs:/mif/sample.mif", StaticApp_Load},
   {NULL, NULL}
};

-------------------------------------------------
```

In the example above, StaticApp_Load is the name of the exported static application's PFNLOAD function, and “sample.mif" is the associated MIF file. If the file "sample.mif" is found in the BREW MIF directory (or as a persistent file - see below), the StaticApp_Load function is associated with the MIF rather than searching for a dynamic code module. This allows the MIF to control the behavior of the application or module code exactly as it would if dynamically downloaded.

**NOTE:** The “sample.mif” letter case must match the letter case of the “sample.mif”string in the AEEConstFile structure.

## Dynamic and constant MIF files

Although OEMs can place MIF files into the file system after flashing the code, a new mechanism was added to allow for files of all types to be compiled and linked into the core software image but appear as if part of the native file system. These rules for constant files apply to MIF files as well. Constant MIF files are located in the flash image of the phone but appear to be part of the file system available to BREW. This mechanism leverages a utility (bin2src.exe) that converts binary files into an equivalent source module (see the *BREW PK Utilities Guide*). The source module is then exported and placed into a table that is leveraged by the BREW to expose the entries as constant files that are available as if part of the native file system. All such files are associated with a path of the format fs:/xxxxxx where xxxx is the directory and file name of the file.

A sample version of a converted MIF file follows:

```
-- samplemif.c ----------------------------------

//*******************************************
//*
//* This file automatically generated using BIN2SRC.
//*
//*******************************************

#include "AEE_OEMFile.h"
#include "AEEStdLib.h"

// Binary data file contents

static const byte gsData[] = {17\

,0,1,0,1,0,4,0,32,0,0,0,32,0,0,0,64,0,0,0,4,0,0,0,84,0,0,0,72,0,0,0,1,0,232
,3,0,0,0,0,0\
,80,0,0,0,0,1,0,0,80,2,0,0,0,2,0,0,80,9,0,0,0,3,0,84,0,0,0,88,0,0,0,128,0,0
,0,148,0,0,0,156\

,0,0,0,86,105,101,119,2,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,
0,0,1,0,0,0,0\

,0,0,0,0,0,0,0,63,16,0,1,0,0,0,0,232,3,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0}
;

// Global structure declaration to be added to OEMFSPersistent.c

const AEEConstFile gSAMPLE_MIF = {"fs:/mif/sample.mif",FALSE,156,0,156,(byte
*)gsData};

-------------------------------------------------
```

**NOTE:** The file above was generated automatically using the bin2src application with the following arguments:

```
            bin2src -ssample.mif -dfs:/mif
```

The resulting source file is called:

```
            samplemif.c
```

After converted to source code form, the file is compiled and linked into the build. To make the file visible to BREW, an associated entry must be added to the file OEMConstFiles.c as follows:

```
-- OEMConstFiles.c ----------------------------

extern AEEConstFile     gBREWAPPMGR_MIF;
extern AEEConstFile     gVIEWAPP_MIF;
extern AEEConstFile  gSAMPLE_MIF;

static const AEEConstFile * gpPersistentFiles[] = {
                                                     &gBREWAPPMGR_MIF,
                                                     &gVIEWAPP_MIF,
                                               &gSAMPLE_MIF,
                                                     NULL};
-------------------------------------------------
```

After it has been added to the build in source form and added to the persistent file list in OEMConstFiles.c, the associated MIF, or any other file, is exposed by BREW as if it had been installed into the file system. These changes allow BREW to enumerate the MIF along with all of the dynamically downloaded MIF and to associate the MIF with statically linked source code that is associated with the specified MIF.

## Important notes

The following is important information that should be noted.

- This mechanism replaces the 1.x-2.x support for static applications or modules. Compiling static applications that have not been converted as outlined above will result in a compilation error for PFNGETINFO;
- This mechanism allows OEMs to leverage either persistent or dynamic MIF files for static code.
- The root name specified for the MIF file must match the name of the MIF in the file system or constant file system table.
- By default, constant MIF files are marked as upgradeable (fixed = FALSE). This allows them to be upgraded over the air if necessary;
- Constant MIF files must be specified with the root directory of fs:/mif in order to be supported correctly on the handset and simulator.

# Appendix A: Using the OEM Extranet

## Using the BREW OEM web sites

### Obtaining an account

You can obtain online information and support by visiting the BREW OEM Extranet at https://brewx.qualcomm.com/oem/home.jsp. You must have an account before you can access the OEM Extranet.

#### To obtain an OEM Extranet account

1. Send an email to brew-oem-support@qualcomm.com and include the following information:

   **a.** Your full name

   **b.** Your email address

   **c.** The company name

   **NOTE:** Before sending this request, verify that your company has an agreement with QIS.

   **d.** Your title

   *A new account is generated, and a notification email is sent to you.*

2. After you receive the notification email with your user name and password, you can log into the OEM Extranet account (https://brewx.qualcomm.com/oem/home.jsp).

3. Change your password.

4. Choose **Information** and verify the information you entered. If you find a discrepancy, contact brew-oem-support@qualcomm.com for correction.

   *The OEM Extranet home page opens.*

## Understanding the customer ID

The customer ID is your QUALCOMM Support Number.

## Obtaining porting support

To report problems or request support for porting BREW on devices, send an email message to brew-oem-support@qualcomm.com.

To obtain platformIDs that reference BREW configuration and related functions, send an email message to brew-oem-support@qualcomm.com.

## Requesting new BREW features

Do you have ideas for features that would make the *BREW OEM Porting Kit for MSM Platforms*, more valuable and useful to you? If so, send us email at brew-request@qualcomm.com. Each request is evaluated, and a member of the New Features Response Team will respond to your email.

## Obtaining release notes

Release notes are produced with each release of the Porting Kit software. Release notes are located on the OEM Extranet under the specific release number.

## Understanding known issues

Outstanding known issues for a specific release are located on the OEM Extranet under the specific release number.

## Obtaining BREW OEM Notes

These notes are designed to provide workarounds for any newly discovered problem, additional information on specific features, and solutions to commonly asked questions. Review the BREW OEM Notes pertinent to the release you are using and verify that all the fixes described in the notes are applied to your build.

Check the OEM Extranet frequently for any updates, as there is no planned BREW OEM Note release schedule. QUALCOMM sends email notifications on new OEM Notes to registered email addressees.

# Appendix B: DMI Compliance

The mobile device must be compatible with all BREW tools, including the AppLoader, BREW Logger, The Grinder®, and Shaker. These tools use the Diagnostic Monitor Interface (DMI) of the QCT DMSS. The DMI used by these tools must be functioning properly on the device, so BREW application developers can transfer applications to the device and test them with the additional tools BREW provides. See DMI compliance command on page 162 for details.

## To verify the DMI compliance of the mobile device

1. Install the BREW Porting Evaluation Kit (PEK) from the OEM Extranet.
2. Launch PEK Studio tool by clicking **Start** > **Programs** > **BREW Porting Evaluation Kit** > **PEK Studio**.
3. Run the Connectivity Test in the PEK Studio tool (see the *BREW Porting Evaluation Kit Online Help* for detailed instructions). If the test fails, refer to the *PEK Test Cases Guide* for instructions for retrying the test. If the test still fails, ensure that the mobile device has the proper implementation for all DMI commands mentioned in DMI compliance command on page 162.
4. Run the PC Interface Test in the PEK Studio tool. Verify that all the test cases passed by either referring to the PEK Studio status window messages or by generating the PC Interface Test Report in the PEK Studio (see the *BREW Porting Evaluation Kit User's Guide* for detailed instructions). If the test fails, see the instructions for retrying the test. If the test still fails, ensure that the mobile device has proper implementation for all the DMI commands mentioned in DMI compliance command on page 162 are properly implemented on the mobile device.

   *If the Connectivity Test and the PC Interface Test continue to fail, and you have any questions on the DMI compliance of your mobile device, contact QUALCOMM or brew-oem-support.*

## DMI compliance command

The following commands must be implemented on the mobile device for it to be DMI-compliant.

| **Command code** | **Operation code** | **Packet name** | **Description** |
|---|---|---|---|
| 0 | N/A | Version Number Request/Response | Gets device software information and other static configuration data. |
| 1 | N/A | ESN Request/Response | Gets the ESN of the device. |
| 12 | N/A | Get Current DMSS Status Request/Response | The status message asks for current DMSS status information |
| 15 | N/A | Logging Mask Request/Response | The status message asks for current DMSS status information. |
| 16 | N/A | Log Request/Response | Retrieves a single queued log item from the DMSS. The DMSS removes the oldest log item (if any), places it in a Log Response Message, and outputs the log item to the DM. |
| 28 | N/A | Diag Version Request/Response | The DM checks the version of the DM/DMSS packet interface in use by the DMSS by sending a Diag Version Request Message. The DMSS responds by sending a Diag Version Response Message containing the version number. If the version used by the DMSS is not the same as that used by the DM, proper interpretation of all packets is not guaranteed. |
| 29 | N/A | Time Stamp Request/Response | Requests the current time in the DMSS. |

| Command code | Operation code | Packet name | Description |
|---|---|---|---|
| 31 | N/A | Message Request/Response | Retrieves the DMSS diagnostic message. As a diagnostic aid, the DMSS records text messages at various points in the execution. The messages provide developers with some insight into the behavior of the DMSS program.<br>Message level (minimum message severity level):<br>• 0000: all messages (MSG_LVL_LOW)<br>• 0001: medium and above (MSG_LVL_MED)<br>• 0002: high and above (MSG_LVL_HIGH)<br>• 0003: error and above (MSG_LVL_ERROR)<br>• 0004: fatal error only (MSG_LVL_FATAL)<br>• 00FF: no messages (MSG_LVL_NONE) |
| 32 | N/A | Device Emulation Keypress Request/Response | Device keypress and the other device conditions are provided through the serial data interface. The request message contains the indicated keypress. This keypress is inserted in the stream of key presses between the device driver and the UI software. |
| 33 | LOCk = 0<br>UNLOCK = 1 | Device Emulation Lock/Unlock Request/Response | Locks and unlocks the handset. To prevent the collision of input, it is recommended that you restrict remote input of devices keystrokes. This is achieved by locking and unlocking the device. On invoking this Lock request, the DMSS locks the device, which prevents user input, and processes the Device Emulation Keypress Messages. On invoking this Unlock request, the DMSS unlocks the handset. |

| Command code | Operation code | Packet name | Description |
|---|---|---|---|
| 41 | Mode = 1 | Mode Change Request/Response Message | Puts the DMSS (device) in digital offline mode. The only exit from the offline mode is through a restart. The device may be power cycled to produce this reset, or the DM (OEM layer) may send a Reset command to reset the device. |
| 70 | N/A | Security Password Request/Response | Sends the security password to the phone to unlock secure operations. Following are the diagnostic packets with secure operations: <br>• Memory Peek Request/ Response <br>• Memory Poke Request Response <br>• NV Memory Peek Request/ Response <br>• NV Memory Poke Request Response |
| 89 | 0 | EFS Operation Request | Create Directory: Creates a specified directory on the device. |
| 89 | 1 | EFS Operation Request | Remove Directory: Removes a specified directory from the device. |
| 89 | 2 | EFS Operation Request | Display Directory List (Enumerate Subdirectories): Displays a list of directories present in the specified directory on the device. |
| 89 | 3 | EFS Operation Request | Display File List (Enumerate Files): Displays a list of files present in the specified directory on the device. |
| 89 | 4 | EFS Operation Request | Read File: Reads the file from the device. |
| 89 | 5 | EFS Operation Request | Write File: Writes to the specified file on the device. |
| 89 | 6 | EFS Operation Request | Remove File: Removes specified file from the device. |

| Command code | Operation code | Packet name | Description |
|---|---|---|---|
| 89 | 7 | EFS Operation Request | Get File/Directory Attributes: Gets file attributes (EFS attributes, creation date, logical size). |
| 89 | 8 | EFS Operation Request | Set File/Directory Attributes: Sets file attributes (Unrestricted, Permanent, Read-Only, System Permanent, or Remote File). |
| 89 | 10 | EFS Operation Request | Iterative Directory List: If this command is enabled, the Display Directory List operation is not supported. The purpose of this operation is the same as that of the Display Directory List operation. |
| 89 | 11 | EFS Operation Request | Iterative File List: If this command is enabled, the Display Directory List operation is not supported. The purpose of this operation is the same as that of the Display File List operation. |
| 89 | 12 | EFS Operation Request | Get Free and Used EFS Space: Get available space and used space on the DMSS (device). |
| 92 | N/A | Configure Communications | This command is used to query available communications speed (bit rates) and to change the bit rate in the DMSS. |
| 93 | N/A | Extended Logging Mask Request/Response | The DM sends the Extended Logging Mask Request Message to the DMSS to collect (or stop collecting) log data of a specified sort. The mask is a list of bits with each bit position specifying a different type of log data. |

# Appendix C: Test Enable Bit Removal

The test enable bit functionality was removed in BREW 3.1. The associated changes described below provide several benefits to speed the development and commercialization of both BREW devices and BREW applications. The major benefits of removing the test enable bit include:

- Enables authenticated BREW developers to develop and test applications on standard BREW devices from Operators' sales stores or other distribution points without risk.
- Enables debugging by OEMs during porting without losing all preloaded or existing applications. By removing the test enable bit, OEMs will be able to more rapidly and conveniently manipulate these device elements without undesirable, adverse affects and thereby be able to more rapidly commercialize BREW devices.
- Enables OEMs and carriers to better test BREW devices in a fully "commercially ready" state. By enabling debug functionality without Test Enable mode on, the device is in a completely accurate state to a commercial device, as opposed to being in a mode that differs from the commercial mode in many respects.

The following table shows what OEMs and Developers previously did using test enable bit and how each task is accomplished in 3.1 with the concept of "test enabled bit" removed.

•

| | **Old behaviour (test enable bit)** | **New behaviour** |
|---|---|---|
| **Test signatures** | BREW Test Signatures are rejected unless test enable bit is on. | BREW Test Signatures is *always* accepted. |
| **Tail-less MIFs** | Tail-less MIFs are rejected unless test enable bit is on. | Tail-less MIFs are accepted if test signature is present. |
| **BREW debug key sequences** | Access to BREW diagnostic functionality (debug keys) is denied unless test enable bit is on. | Access to BREW debugging functionality is always granted. Debug key sequences are improved, made more secure and capable (see SDK User's Guide. |
| **Testing BREW-directed SMS** | BREW-directed SMS  must start with "//" when Test Enable bit is off.  BREW directed SMS must ***merely contain*** "//" when Test Enable bit is on. | Testing BREW-Directed SMS (relaxing parsing requirement) governed by new Debug Key sequence (see SDK User's Guide). |
| **dlservers.dat** | IDownload APIs for getting servers from dlservers.dat and setting the download server disabled unless Test Enable bit is on. | IDownload APIs unrestricted (guarded by PL_DOWNLOAD instead |

# Index