QUALCOMM

Binary Runtime Environment for Wireless®

BREW® 3.1.5 OEM Porting Guide for MSM Platforms

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

This documentation was written for use with the BREW® Porting Kit for Windows, software version 3.1.5. This documentation and the BREW Porting Kit software described in it are copyrighted, with all rights reserved. This documentation and the BREW Porting Kit software may not be copied, except as otherwise provided in your software license or as expressly permitted in writing by QUALCOMM Incorporated.

BREW® 3.1.5 OEM Porting Guide for MSM Platforms

80-D7960-1
March 18, 2008

Contents

BREW OEM Porting Guide for MSM Platforms

The BREW® OEM Porting Guide for MSM Platforms 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®). It includes a Porting Kit Overview, a Quick Start section with a summary of each porting step, detailed sections for each of the porting steps, and advanced topic sections for additional information. References are made to the advanced topics throughout the porting guide or to other documents that you may find helpful. A list of acronyms and terms is provided in Appendix A on page 202.

BREW OEM Support

For any support issues, please open a service request at http://support.cdmatech.com or email to support.cdmatech@qualcomm.com.

Overview

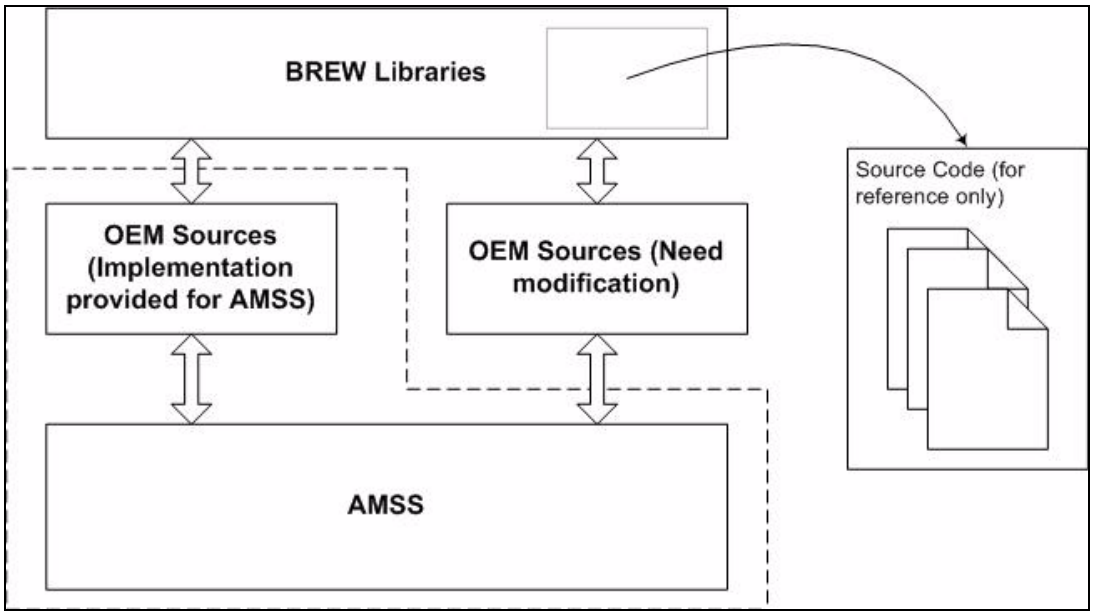
BREW provides a layer on top of the chip system software. This makes the device's functionality available to the application without requiring the developer to have the chip system source code or even a direct relationship with the device manufacturer.

The OEM Porting Kit tells OEMs how to integrate BREW with their device. 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



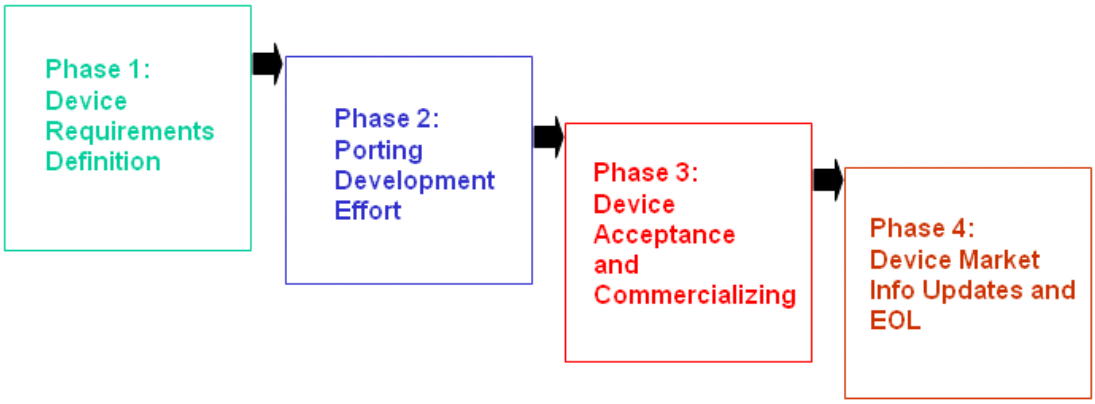
Porting Kit Components

Component	Description
BREW library files (BREW*.lib)	The library files are available for linking with the device software using the ARM Developer Suite. 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 files 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.



The following is a description of the phases:

Phase 1	The operator communicates their 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.

This section contains procedures that will 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.

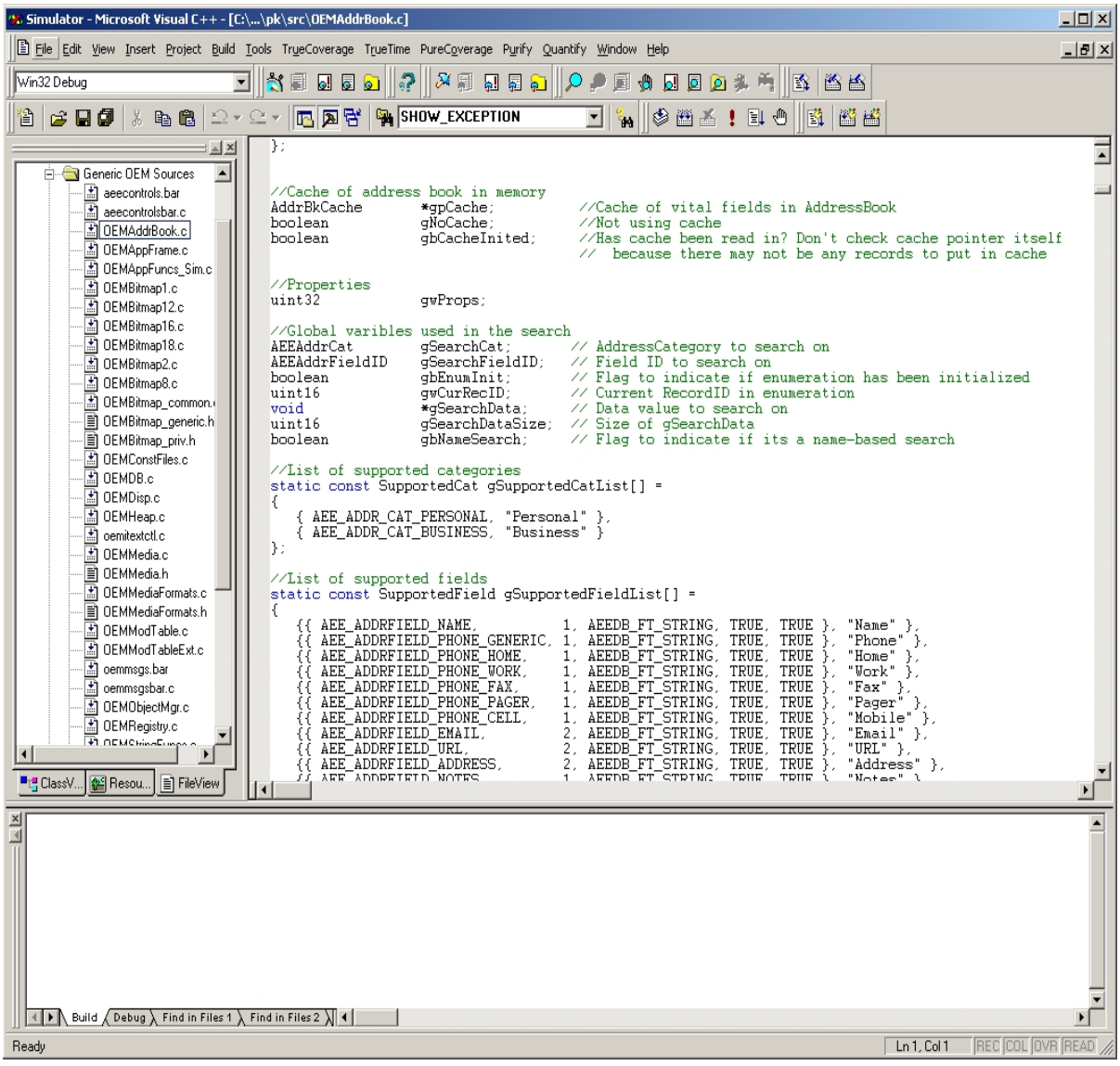
Using the BREW Simulator

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. The simulator files are shipped so that OEMs can test changes made to OEMFiles, the BREW App Manager, or custom static BREW apps and extensions in a Windows environment. OEMs who have their own simulators can link in BREWWin.lib.



Step 1. Initialize BREW

NOTE: For consistency, the terms “task” and “signal” are used throughout this document. “Task” is synonymous with “thread,” and “signal” refers to any inter-task communication construct.

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. QUALCOMM recommends a stack size of at least 16kB for the task in which BREW runs..

   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. 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.
5. 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 Initialize 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 device events and notifications to BREW using the following macros:

   - AEE_SEND_FLIP_EVT (p)
   - AEE_SEND_KEYGUARD_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.

3. Send pen events using the AEE_Event() function if your device is touch-screen capable. The event codes can be one of the following:

   a. AEE_Event (EVT_PEN_DOWN, 0, dwPos) where the x coordinate is expressed as a signed integer in the upper 16 bits of dwPos, and y is a signed integer in the lower 16 bits. This event is sent when the pen comes in contact with the screen.

   b. AEE_Event (EVT_PEN_MOVE, 0, dwPos) where the x coordinate is expressed as a signed integer in the upper 16 bits of dwPos, and y is a signed integer in the lower 16 bits. This event is sent when the pen moves on the screen while maintaining contact with the screen.

   c. AEE_Event (EVT_PEN_UP, 0, dwPos) where the x coordinate is expressed as a signed integer in the upper 16 bits of dwPos and y is a signed integer in the lower 16 bits. This event is sent when the pen leaves the screen.

4. Send joystick events using the AEE_Event() function if your device supports joystick functions. Call AEE_Event (EVT_JOYSTICK_POS, 0, dwPos) where the x coordinate is expressed as a signed integer in the upper 16 bits of dwPos, and y is a signed integer in the lower 16 bits.

See Set up event handling 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 Bitmaps on page 164.

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 Configure the Device on page 59.

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. See the BREW Services sections for more information on enabling specific BREW features, and also 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 Downloading BREW applications on page 88.

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.

Refer to BREW and native applications on page 180 and .

Step 9. 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 10. Create MCF directories

1. Copy all the MIF (*.mif) files from the "/pk/mcf-mifs" folder to the BREW MIF directory on the device (AEEFS_MIF_DIR). Assuming that AEEFS_MIF_DIR is set to /brew/mif/, it should look like the following:
   a. /brew/mif/10888.mif
   b. /brew/mif/10889.mif
   c. /brew/mif/10890.mif
   d. /brew/mif/10891.mif
   e. /brew/mif/10892.mif
   f. /brew/mif/18067.mif
2. Create corresponding module directories in the BREW MOD directory on the device (AEEFS_MOD_DIR). Assuming that AEEFS_MOD_DIR is set to /brew/mod/. It should look like the following:
   a. /brew/mod/10888
   b. /brew/mod/10889
   c. /brew/mod/10890
   d. /brew/mod/10891
   e. /brew/mod/10892
   f. /brew/mod/18067

Step 11. Perform testing

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

See xxxDave's Requirements Check List, and the *BREW Porting Evaluation Kit (PEK) User's Guide*.

Initialize BREW

This section discusses the steps involved in initializing and terminating BREW and calling into BREW's dispatcher routine.

NOTE: For consistency, the terms "task" and "signal" are used throughout this document. "Task" is synonymous with "thread," and "signal" refers to any inter-task communication construct.

Identify a task to run BREW

Before beginning the integration of BREW onto your device, you need to decide the task in which to run BREW. Most BREW code will execute in the context of the task you choose. You may run BREW in a task that already exists in your software or define a new task specifically for BREW. QUALCOMM recommends that you run BREW in the pre-existing task that handles your User Interface functionality. If you decide to create a new task for BREW, QUALCOMM recommends that the priority of that task be immediately below your User Interface task and above the file system and sleep tasks.

Since all BREW applications run in the context of the task in which BREW is running, you need to make sure the task's stack size is sufficient. QUALCOMM recommends a stack size of at least 16Kbytes for the task in which BREW runs.

NOTE: For QUALCOMM AMSS releases with BREW pre-integrated, BREW runs in the UI task. See the AMSS documentation for details on the task priorities and stack sizes.

Once you have chosen the task in which to run BREW, you need to add calls into the key BREW functions. How and where to add calls into the key BREW functions is explained below.

Add calls to initialize and terminate BREW

BREW provides two key functions for initializing and terminating BREW, AEE_Init() and AEE_Exit(). Refer to the *OEM API Reference Online Help* for more details on BREW functions.

AEE_Init(), BREW's initialization function, must be called from the task in which BREW will run as soon as possible during device power up. During AEE_Init(), BREW will malloc and initialize its internal data structures, initialize any extensions that require initialization on startup, determine the device hardware and Subscriber IDs (SIDs), scan the file system for any dynamic modules, and schedule the Autostart application to start, if any.

AEE_Exit() must be called from the task in which BREW is running. AEE_Exit() is BREW's shut-down routine and must be called during device power down. Failing to call AEE_Exit() when the device powers down may result in the device entering into Safe Mode. Refer to the advanced section for more information on the BREW Safe Mode.

NOTE: BREW should always be initialized while the device is powered on. It is required that BREW be initialized (via AEE_Init()) on power up and exited (via AEE_Exit()) when the device is powered down since BREW provides services to applications even when the applications are not in the running state. For example, BREW applications may set alarms to wake up at a certain time. However, these services are only available while BREW is initialized.

Add calls to the BREW Dispatcher routine

All BREW applet code is executed from BREW's dispatcher routine, AEE_Dispatch(). AEE_Dispatch() is responsible for maintaining and expiring any applet-set timers, dispatching any pending events to applets and for running the BREW applet code.

This routine must be called from the task in which you decide to run BREW, and it must be called each time BREW signals to the OEM Layer that it needs to run. BREW will request the OEM Layer to call the AEE_Dispatch() routine by calling OEMOS_SignalDispatch(). When BREW calls OEMOS_SignalDispatch(), the OEM Layer must schedule the task in which BREW is running to call AEE_Dispatch().

For QUALCOMM AMSS releases with BREW pre-integrated, AEE_APP_SIG is a UI task signal defined specifically for BREW. The UI task loop looks something like the following:

```
for( ;; ) /* Wait on REX signals, repeat forever */
{
   sigs = rex_wait(
          /* … other UI Task Signals … */
          | AEE_APP_SIG
          );

   /* … handle other UI task signals… */
   if (sigs & AEE_APP_SIG)
```

```
    {
       rex_clr_sigs(rex_self(), AEE_APP_SIG);
       OEMDisableSleep(); //OEM Provided disable sleep routine
       ret = AEE_Dispatch();

       if (retval==0){
          //No BREW short term timers and no BREW callbacks pending..ok to
          sleep now
         OEMEnableSleep(); //OEM Provided enable sleep routine
        }else{
         OEMDisableSleep();//OEM Provided disable sleep routine
        }
    }
}
```

In the above implementation, rex_wait() is called to wait on a number of signals that may be of interest to the UI task. For simplicity, only the BREW signal (AEE_APP_SIG) is shown here. rex_wait() will return whenever one of the signals that the task is waiting on is set. In the event that AEE_APP_SIG is set, the code above will clear the signal from the TCB and call AEE_Dispatch().

In addition to dispatching BREW events the code above also manages the UI tasks sleep vote. The return value of AEE_Dispatch() dictates whether it is ok for the device to go to sleep or not when outside of BREW. If any value other than 0 is returned from AEE_Dispatch(), it must be ensured that the device does not go into the sleep mode. Also, it must be ensured that sleep mode is not entered prematurely during the processing of AEE_Dispatch(). There could be cases where the processing of AEE_Dispatch() could take a significant amount of time which could cause the sleep to kick in and drastically reduce the end user experience. To avoid this scenario, disable sleep before invoking the AEE_Dispatch() routine and on return disable / enable sleep based on the return value of AEE_Dispatch().

For More Information

See the *OEM API Reference Online* Help for more information on the following topics:

- AEE_Init()
- AEE_Exit()
- AEE_Dispatch()
- OEM_GetConfig(CFGI_SLEEP_TIMER_RESOLUTION)
- OEM_GetConfig(CFGI_MAX_DISPATCH_TIME)

- OEM_GetConfig(CFGI_MIN_IDLE_TIME)

Set up event handling

This section contains information about sending key events and other events to BREW. When an event is sent to BREW synchronously using ISHELL_SendEvent() or AEE_Event(), the return value indicates whether the event was handled by BREW or a BREW application. However, when an event is sent asynchronously using ISHELL_PostEvent(), the return value indicates whether the event has been queued, but not necessarily handled.

Sending key events

It is extremely important to send key events to BREW at all times. Although BREW key events are sent with AEE_Event(), they are handled asynchronously. The return value of the AEE_Event() function with EVT_KEY, EVT_KEY_PRESS or EVT_KEY_RELEASE indicates whether the key event has been queued.

In scenarios where 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.

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. BREW starts an internal timer as soon as it receives the first EVT_KEY following EVT_KEY_PRESS. 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: It is important to remember to implement OEM_GetDeviceInfoEx() for the AEE_DEVICEITEM_KEY_SUPPORT and AEE_DEVICESTATE_KEY_INFO parameters. See the *OEM API Reference Online Help* for more information.

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)

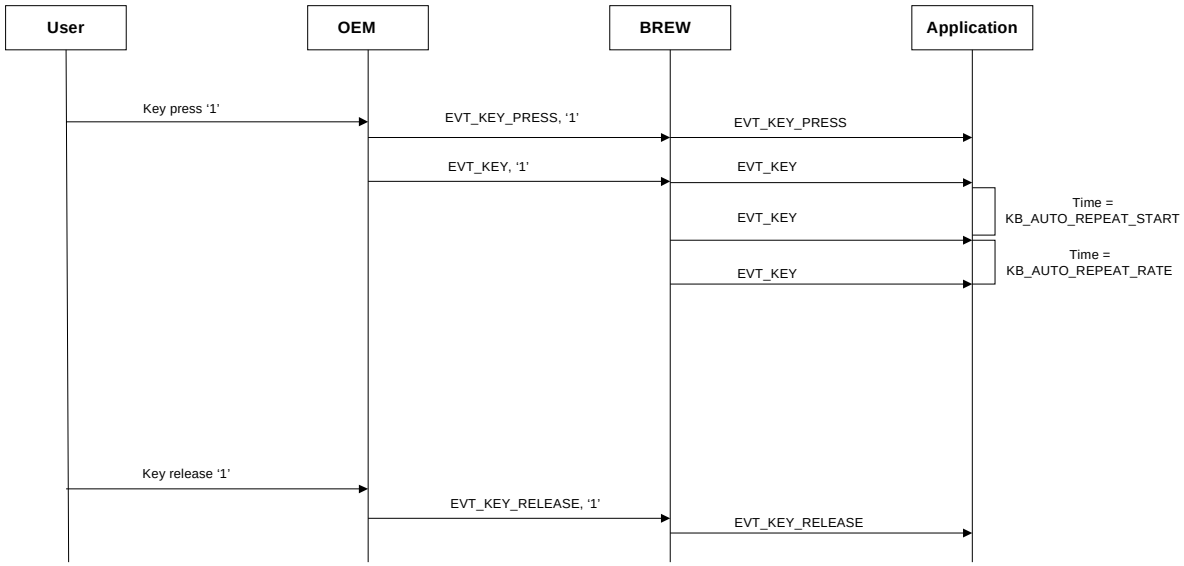
  The timer sets to KB_AUTOREPEAT_RATE or the OEM-defined value if it is nonzero.

  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



Key modifiers

There are several key modifiers defined for BREW, in AEEVCodes.h, which must be passed along with the BREW Virtual Key codes. This value is passed as the dwParam value (third parameter of the type uint32) in the AEE_Event() function. For every key modifier you detect, you will need to perform a bit-wise OR operation to send multiple key modifiers if there is more than one.

NOTE: When there are key modifiers, you cannot use the AEE_Key() macro. Instead, you must use AEE_Event(). For example, if the key 1 was pressed along with the Left Shift key and the Left Alt key, you must make sure the two modifiers are included in the key event as shown below:

```
AEE_Event(EVT_KEY, AVK_1, (KB_LSHIFT | KB_LALT));
```

NOTE: When key modifiers are used, you still must generate key events, for example, AEE_Key, AEE_KeyPressed and AEE_KeyReleased for each of the modifier keys.

Capturing key events

Prior to BREW 3.1, key events could only be received by the top-visible BREW application. In BREW 3.1 privileged BREW applications can receive all key events prior to being sent to any other BREW application. All key events are sent to the privileged BREW Application even when it is not the top-visible application. The privileged BREW application can be created by following the steps below when creating its MIF file.

To create the privileged BREW application

1. Use the 3.x MIF editor, and under the Applets tab, click **Notifications, Flags, Settings**.... In the new window that pops up, check the phone box under the Flags section. This sets AFLAG_PHONE for the application.
2. Enable system privilege for the application. This is done by selecting the Privileges tab and checking System privilege in the MIF editor.

Once this is done, AFLAG_PHONE privileged application will get EVT_KEY_HOOK, EVT_KEY_HOOK_PRESS and EVT_KEY_HOOK_RELEASE events for every key pressed. The AVK_ code for the key will be in the wParam parameter of the event handler function of the application.

The AFLAG_PHONE privileged application has consumed the key event if it returns TRUE to the key hook events. If the key hook event is consumed, the key event is not passed to the current top-visible application. This provides a way for privileged applications to provide custom handling for specific key events such as AVK_END.

NOTE: There should be a very small number (preferably just one) of applications on the device that have AFLAG_PHONE set. If not, it can cause slowness on the device since every key will be first sent to each of the currently running AFLAG_PHONE applications prior to sending to the top-visible application.

In BREW 3.1, applications can register for specific key events to be received even when they are running in the background. Registering for a specific key event is achieved by calling ISHELL_RegisterNotify() with NMASK_SHELL_KEY as the lower 16bits and AVK code of the desired key as the upper 16bits of dwMask parameter.

To register for all key events, NOTIFIER_VAL_ANY should be used for the AVK code. An application can also register for specific key events in the MIF of the application instead of using the ISHELL_RegisterNotify() API. For each registered key, the application will receive EVT_NOTIFY event in its event handler with wParam as the AVK code and dwParam of the type AEENotify. The pData member inside this AEENotify structure is of type NotifyKeyEvent. For more information on AEENotify and NotifyKeyEvent, refer to the *BREW OEM API Reference Online Help*.

The key events are always first sent to any privileged applications that have registered for key hook events. If any of these privileged applications handle the key event but do not consume it, the clsHandled member of NotifyKeyEvent received on EVT_NOTIFY is set to the class of the privileged application. If none of the applications, which have registered for key hook events, handle the key event, then the key event is sent to the current top-visible application. If that top-visible application handles the key event, then clsHandled is set to the classID of the top-visible application. The top-visible application cannot consume the key event and prevent it from being delivered to registered applications.

An application that has registered for key notification events can prevent the notification from going to other applications by consuming the notification. The way it does this is before returning from the EVT_NOTIFY handler, it needs to set the “st” field of dwParam to NSTAT_STOP.

```
case EVT_NOTIFY:
   (AEENotify *)wp = (AEENotify *)dwParam;
   wp->st = NSTAT_STOP;
   return TRUE;
```

If multiple applications have registered for key press notification (not key hooks), then the order in which the notifications are sent depends on the order in which the applications registered with brew. The notification deliver order is the reverse of the registration order. If A1 registered first followed by A2, the notification is first sent to A2 and then to A1. If the registration for the notifications is done through the MIF, the registration order cannot be guaranteed.

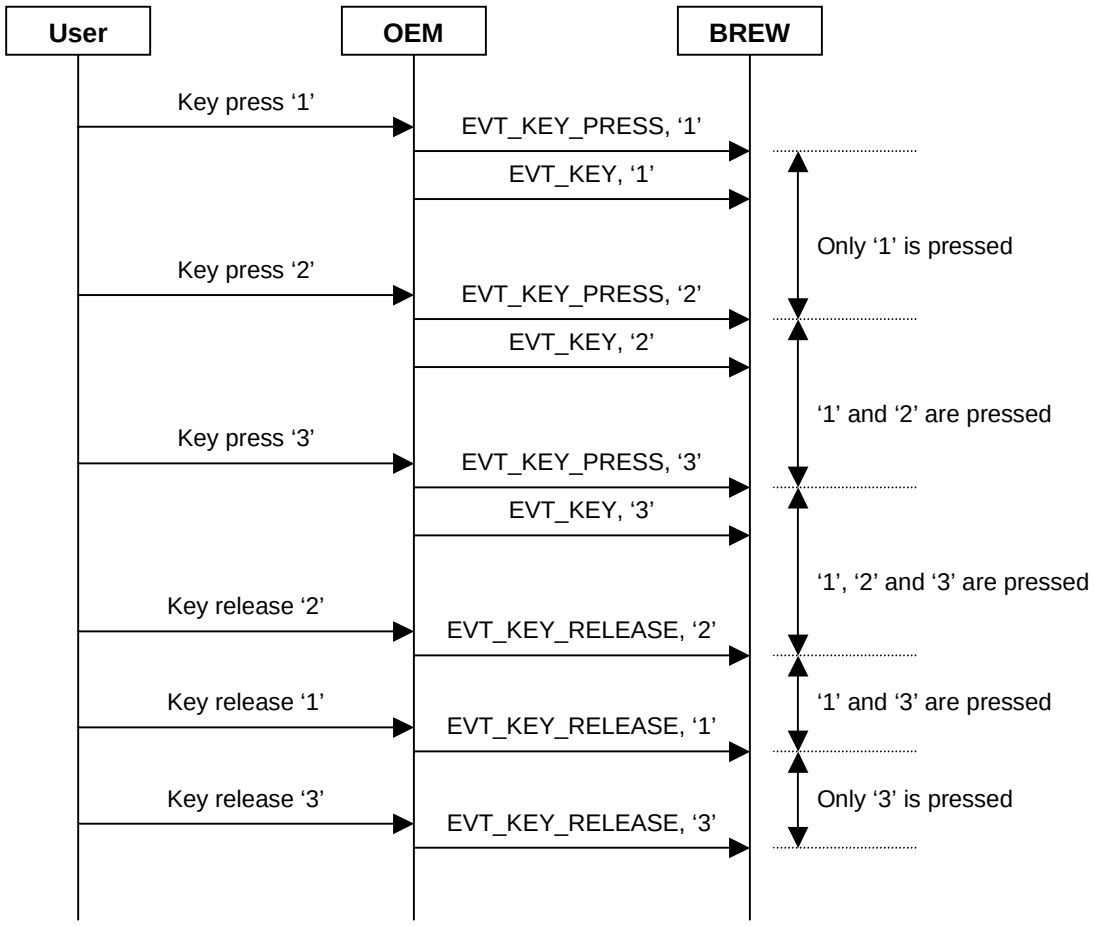
Multiple keypress support

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.

Porting Extended keypads

The steps needed by OEMs to port extended keypad formats such as QWERTY on BREW are listed. To create a standard approach for extended keypads, these have been introduced:

1. Standard key events to be sent to BREW for extended keys.
2. A mechanism for interpreting the key code combinations (IKeysMapping interface)
3. An interface for setting and getting the state of sticky keys (IKeysConfig)
4. FASTAP integration

Standardizing key events

To standardize key events

1. All extended keys including alphabet keys, special symbols and modifiers such as the SHIFT key will generate `EVT_KEY_PRESS`, `EVT_KEY`, `EVT_KEY_RELEASE` events.
2. New virtual key codes from `AVK_A` to `AVK_Z` are defined for alphabet keys from A to Z. The key codes have a numeric value equal to the Unicode value of the corresponding lower case letter. For example, `AVK_A` has the value 97 decimal, which is the Unicode value for lowercase a.
3. Alphabet keys from A to Z will generate EVT_KEY events with wParam equal to the Unicode value of the lower case letter. As an example, pressing the Q key will generate `EVT_KEY_PRESS, EVT_KEY, EVT_KEY_RELEASE` events with `wParam` of `AVK_Q` or 113 decimal.
4. OEMs should not interpret key code combinations – they will always send EVT_KEY events with the same `wParam`, and `dwParam` will contain modifiers. This means that if the user presses SHIFT and A together, the OEM will send EVT_KEY events with `wParam` still equal to the Unicode value of lowercase a (`AVK_A` , for example, 97 decimal), and `dwParam` with `KB_LSHIFT` or `KB_RSHIFT`, as appropriate.
5. Extra modifiers, `KB_SYMBOL`, `KB_FUNCTION`, are now defined to support the SYMBOL and the FUNCTION key. These keys are often used to overload extended keys. For example, pressing [Symbol] + [A] can mean the character #.

To support this standardization, OEMs are expected to call `AEE_Event`, with the corresponding values of `wParam` and `dwParam` depending on the key(s) pressed. OEMs should check the key modifier state (such as Shift key state), and send the state data together with the key value to BREW. For example, if the user presses [Shift] + [A], this sample sequence of calls is expected:

```
/* User pressed left SHIFT */
AEE_Event(EVT_KEY_PRESS, AVK_LSHIFT, 0);
AEE_Event(EVT_KEY, AVK_LSHIFT, 0);

/* User pressed A (and is still holding SHIFT) */
AEE_Event(EVT_KEY_PRESS, AVK_A, KB_LSHIFT);
AEE_Event(EVT_KEY, AVK_A, KB_LSHIFT);

/* User released A */
AEE_Event(EVT_KEY_RELEASE, AVK_A, KB_LSHIFT);

/* User released left SHIFT */
AEE_Event(EVT_KEY_RELEASE, AVK_LSHIFT, 0);
```

Defining a mechanism for interpretation of key codes

Since extended keypad layouts are not standard, it is possible that a key combination may have a different interpretation on different handsets. For example, [Symbol] + [A] can mean # on one handset and * on another. Since applications are expected to be generic and not be required to know the layout of the particular keypad they are running on, this entails a mechanism where the application can find out the interpretation of a particular key combination.

For supporting this mechanism, a new interface IKeysMapping is being introduced. Applications can call IKEYSMAPPING_GetMapping to get the mapping of a virtual key code and a modifier flags combination. See the example of how an application would use this interface.

```
boolean App_HandleEvent(…)
{
      switch( eCode )
      case EVT_KEY:
       if ( dwParam )
      {
           AECHAR mapping;

        if (  IKEYSMAPPING_GetMapping(pMe->pKeyMapping,
                wParam, dwParam, &mapping) != SUCCESS )
            return FALSE;
```

```
                //Use mapping
                //…
    }

}
```

The IKeysMapping interface is implemented in the OEM layer at pk\src\OEMKeysMapping.c. OEMs are not required to change this file except to define the location of the “map.csv” file.

To support this, OEMs are expected to provide the mapping of the AVK value and modifiers combination to this interface. This mapping is expected to be provided in the form of a mappings file, KeysMapping.ini.

KeysMapping.ini is a new file required from the OEM. This file should be submitted as part of the device pack, and should also be preloaded on the device along with the interface. The interface will parse this file and create a mappings table that it will use to return the mapping.

The KeysMapping.ini file should define all the keys with modifiers, including all 26 Latin characters (lower case to upper case) and contain one line per mapping.

NOTE: OEMs should have mappings for both left and right modifiers, for example, AVK_A + KB_LSHIFT = A and AVK_A + KB_RSHIFT = A.

The format of each line in this file is of the form [AVK_VALUE] + [MODIFIERS] = [MAPPING]. Some examples are shown:

AVK_A + KB_LSHIFT = ‘A’

AVK_A + KB_RSHIFT = ‘A’ // Need to specify each mapping

AVK_A + KB_SYMBOL = ‘#’

AVK_A + KB_LALT | KB_LSHIFT = 0xA9 // Pressing ALT + SHIFT + A means ©

The mappings file is free-form, and the format supports mappings to be entered in hexadecimal for hard to type characters. Also C++ style single line comments are supported.

Once the file is generated, it needs to be parsed into a form suitable for the IKeysMapping interface. For this, the keysmappingparser tool present in pk\utils, can be used. It is a Windows utility that will generate a map.csv file, which is a parsed version of KeysMapping.ini. The map.csv file should be loaded onto the handset by the OEM, and the OEM then changes the“#define MAPPINGS_FILE to point to the actual location of map.csv on the handset. This location can be anywhere on the handset. The #define MAPPINGS_FILE must use the OEM path format (for example, /brew/usr/key/map.csv), not the BREW path format, fs:/....

Porting instructions

This section outlines all the porting steps required to support this feature.

NOTE: You need to enable FEATURE_KEYS_MAPPING in pk\inc\OEMFeatures.h to support this feature.

To send key events

1. Send key events to BREW as described in the Key Events section. For each alphabet key or modifier, call AEE_Event with EVT_KEY_PRESS, EVENT_KEY, and EVT_KEY_RELEASE.

   NOTE: Note that the standard AEE_KeyPress, AEE_Key, AEE_KeyRelease macros cannot be used here as you may need to specify the dwParam modifier to the key event.

2. For character keys on extended keypads that do not have standard BREW AVK codes, such as keys for single quotes or commas, make sure the parameter to the AEE_Key macros is the Unicode value of the non-shift character on that key (for example, the character that is meant when the key is pressed by itself without any modifiers).

   - For example, if the OEM has a key that has two characters on it, semi-colon (;) and colon (:).
   - Semi-colon is the main character on the key, for example, it is the character meant when this key is pressed by itself. So that when this key is pressed, the parameter to the AEE_Key macros should be the Unicode value of the semi-colon character (59 decimal).
   - The same rules apply if the key is pressed with the shift key. In this case, the application will get EVT_KEY events with wParam = 59 and dwParam containing KB_LSHIFT or KB_RSHIFT.
   - Then, when the application calls IKEYSMAPPING_GetMapping, it should get the Unicode value of colon (58 decimal)
   - To support this, IKeysMapping.ini should have mappings like this:

     ';' + KB_LSHIFT = ':' // Semicolon key is overloaded

     ';' + KB_RSHIFT = ':'

To parse KeysMapping.ini

Follow these steps to create a mappings file for `IKeysMapping`:

1. Create the KeysMapping.ini file following the guidelines in Defining a mechanism for interpretation of key codes on page 36.
2. Use the PK\Utils\KeysMappingparser.exe utility to generate the map.csv file. See Example usage of the parsekeysmapping utility on page 39.
3. The utility requires the BREW SDK AEEVCodes.h file to create the map.csv file. It looks for this file using the BREWDIR environment variable. You should set BREWDIR to the SDK directory of the BREW version for your handset. It is also possible to specify the AEEVCodes.h file that should be used via the `-vc` command line switch.

Example usage of the parsekeysmapping utility

```
C:\BREW\PK\utils> keysmappingparser
C:\BREW\PK\utils> keysmappingparser -k C:\mapping\KeysMapping.ini
```

```
C:\BREW\PK\utils> keysmappingparser -k C:\mapping\km.ini -vc
AEEVCodes.h
```

The first example usage (without any parameters) will look for KeysMapping.ini in the current folder, and will look for AEEVCodes.h in the BREWDIR\inc folder.

The second example will look for KeysMapping.ini in the specified location, and will look for AEEVCodes.h in the BREWDIR\inc folder.

The third example will look for “c:\mapping\km.ini” as the Keys mapping file, and will look for AEEVCodes.h in the current folder (the folder in which the utility is run).

The end result of this step is the “map.csv” file that is generated by the utility. The file is generated in the directory from which the tool is run.

IKeysConfig

A new interface, IKeysConfig, is now defined in BREW to be able to query and set the state of sticky keys on a handset.

NOTE: OEMs are still expected to include sticky keys as modifiers in EVT_KEY events as explained in Porting Extended keypads on page 35. IKeysConfig is not intended to replace that requirement .

This interface can be used by applications to get and set the state of sticky keys, for example, keys such as CAPS LOCK, SCROLL LOCK, and NUM LOCK, which turn on when pressed once, and then get added as a modifier to every key event until they are turned off.

NOTE: Note that OEMKeysMapping.c contains the implementations of two interfaces, IKeysMapping and IKeysConfig.

This interface does not have a reference implementation - the implementation in OEMKeysMapping.c has stubs, which you need to implement if you support this interface.

To port IKeysConfig:

1. Uncomment these lines to your OEMFeatures.h file:

   ```
   #define FEATURE_EXTENDED_KEYS
   #define FEATURE_KEYS_CONFIG
   ```

2. These functions need to be implemented in OEMKeysMapping.c. They are stubs in the provided reference file.

```
static int IKeysConfig_GetStickyKeys(IKeysConfig *pme, uint32
*pdwKeys)
```

This function should return a mask of available sticky keys on the handset. For example, if you have Caps Lock and Num Lock sticky keys, then set *pdwKeys to KB_CAPSLOCK | KB_NUMLOCK.

```
static int IKeysConfig_GetKeyState(IKeysConfig *pme, uint32 dwKey,
boolean *pbState)
```

This function should return the current state of the sticky key being queried. Return TRUE in *pbState if sticky key "dwKey" (example KB_CAPSLOCK) is currently active. Return FALSE otherwise. If dwKey passed in is not a valid sticky key, return EFAILED from the function.

```
static int IKeysConfig_SetKeyState(IKeysConfig *pme, uint32 dwKey,
boolean bState)
```

This function should set the state of the sticky key passed in dwKey to the requested state (If bState is TRUE, turn the sticky key ON, else turn it OFF). Return EFAILED if the operation is not supported for the passed-in key or if the operation fails for some other reason.

FASTAP Support

Integration of support for FASTAP is in accordance with expanding keypad support for the QWERTY.

Fastap is a keypad technology and driver that can detect and surmise what keys are pressed. Specifically Fastap has the concept of raised keys and low keys, alternately placed on the keypad. What this does is that high keys or raised keys can be easily pressed without error, while the user can easily hit some high keys while trying to press the low keys.

Fastap has three algorithms in their driver to correct this possibility:

- Low keys always take priority over high keys.For example, if a low key and one or two high keys are pressed at the same time, the driver interprets that as a low key press.
- If a user presses diagonal high keys (each low key is surrounded by 4 high keys in a square), then this is also interpreted by the driver as a low key press, even though the low key was not actually hit.

- If a high key is pressed and a low key is pressed while the high key is still pressed, the keypad interprets this as a low key press.

It is this third point that cannot be integrated with BREW, because if the delay is sufficiently small (less than 160ms), then the driver just sends one event – the event for the low key (the high key press is ignored). However, if the delay is more than 160ms (a high key is pressed and a low key is pressed with it, but the low key was pressed late), the driver will send out the high key press after 160 ms, and then once it sees the low key press, it will send out a key delete event for the high key, followed by the key down event for the low key. BREW does not have a concept of a key delete event. So the approach for Fastap is:

1. The interaction between the driver and the OEM porting layer will be the same as it is between the HS and UI tasks for normal keypads (as an example). The OEM layer will get key_down and key_up events and will call AEE_Event for EVT_KEY_PRESS, EVT_KEY, EVT_KEY_RELEASE as before. For QWERTY keys, the mapping will be as decided in the new standard approach described above.
2. For the key cancel event, there is no equivalent. So, BREW will end up giving two events to the app, the EVT_KEY events for the (mistaken) high key followed by the EVT_KEY events for the low key that was actually intended. So, the user will end up thinking that they hit the wrong key (which they did, just that Fastap correction could not be applied), and will correct the key.

   NOTE: This is expected to be a rare occurrence.

3. For the wrong key, Fastap will give key_down followed by key_delete (not key_up). So the key_delete will have to be interpreted as a key_release and an EVT_KEY_RELEASE will have to be sent for the high key. Therefore, the OEM should call AEE_Event with EVT_KEY_RELEASE for the key_delete event from FASTAP driver.

Handling keyguard, flip, screen rotation, and headset

The following are guidelines on how handsets that have a clamshell, keyguard, or support screen rotation and headsets in BREW should behave.

When a clamtype device is open or closed, OEMs should:

- *Not* suspend or resume BREW; that is, do *not* call AEE_Suspend() or AEE_Resume().
- *Not* terminate BREW applications; that is, do *not* call ISHELL_CloseApplet().
- Send an EVT_FLIP event to BREW by defining the following macro:

```
#define AEE_SEND_FLIP_EVT_EX(p,b) {\
b = AEE_Event (EVT_FLIP, (p)->wParam, (p)->dwParam); \
(void) AEE_Notify (AEECLSID_DEVICENOTIFIER,
NMASK_DEVICENOTIFIER_FLIP, p);}
```

  Where p is a pointer to a variable of the type AEEDeviceNotify and b is a Boolean variable indicating whether this event has been handled by the top visible BREW application. See the *BREW API Reference Online Help* for more information.

  - If the value b, after calling the above macro is FALSE, you must call ISHELL_CloseApplet(AEE_GetShell(), TRUE.
- Not return FALSE to OEM_Notify() for the OEMNTF_APP_START notification type, when the flip is in the closed state. BREW applications must be allowed to start when the flip is closed, regardless of whether the application was started due to an alarm, an incoming SMS, or any other external events.
- Continue to call the BREW dispatcher, AEE_Dispatch(), when the BREW signal is set while the flip is closed.
- Support all BREW interfaces while the flip is closed; for example, sending or receiving SMS, playing tones, originating data calls, drawing to the secondary LCD, and so forth.
- Update the LCD when the flip is opened. OEMs can ignore the display update function while the flip is closed.
- Dispatch BREW-directed SMS messages to the corresponding BREW application when the flip is in the closed state.

When the keyguard on a device is activated, OEMs should:

- Send an EVT_KEYGUARD event to BREW using the macro:
  `AEE_SEND_KEYGUARD_EVT (pAEEDeviceNotify)`

When the screen is rotated on a device, OEMs should:

- Send an EVT_SCR_ROTATE event to BREW using the macro
  `AEE_SEND_SCRROTATE_EVT (pAEEDeviceNotify)`

When a headset is plugged in or removed on a device, OEMs should:

- Send an EVT_HEADSET event to BREW using the macro:

```
AEE_SEND_HEADSET_EVT (pAEEDeviceNotify)
```

NOTE: The AEE_SEND_FLIP_EVT, AEE_SEND_FLIP_EVT_EX, AEE_SEND_KEYGUARD_EVT, AEE_SEND_SCRROTATE_EVT, and AEE_SEND_HEADSET_EVT macros should only be called from the same task in which BREW runs.

NOTE: In addition to sending these events and notifications, you must also implement the IFlip interface. Additionally, you must set all the right values for the OEM_GetDeviceInfoEx() with the following parameters:

- AEE_DEVICESTATE_FLIP_OPEN
- AEE_DEVICESTATE_KEYGUARD_ON
- AEE_DEVICESTATE_HEADPHONE_ON
- AEE_DEVICESTATE_SCR_ORIENTATION

See the *BREW API Reference Online Help* for more information.

Sending pointer events

Starting with BREW Porting Kit (MSM) 3.1.4SP01, pen events (EVT_PEN_DOWN, EVT_PEN_MOVE, EVT_PEN_UP and EVT_PEN_STALE_MOVE) will be marked as deprecated in favor of more generic pointer events. BREW will support EVT_POINTER_DOWN, EVT_POINTER_MOVE, EVT_POINTER_UP and EVT_POINTER_STALE_MOVE events generated by pointing devices such as a stylus, touch-screen, mouse etc. These events allow the OEM layer to notify BREW and applications about pointing device activity such as putting the pointer down or lifting it up on a capable screen or moving the pointing device across the screen.

The new events and their parameters are listed below:

Pointer events generated by OEMs	Timing
EVT_POINTER_DOWN	Sent when the pointing device is put down on the capable screen.

Pointer events generated by OEMs	Timing
EVT_POINTER_MOVE	Sent when the pointing device is moved across the capable screen while remaining in touch with the screen.
EVT_POINTER_UP	Sent when the pointing device is lifted up from the capable screen.

Pointer events generated by BREW	Timing
EVT_POINTER_STALE_MOVE	This pointer event must not be sent by OEMs. BREW will send this event to the apps when it sees an EVT_POINTER_MOVE event, and if there is a move near a pending EVT_POINTER_MOVE event in the event queue.

IF there are 50 pointer move events already in the queue and more move events are received, the events already in the queue will be trimmed starting with the second event and increasing up to the 50 in order to create equally spaced move events. This will cycle until the BREW app starts consuming events from the queue. This way BREW can accept the newer move events, while still maintaining a fairly accurate trace of the older move event.

The table below contains the pointer event parameters to be passed to AEE_Event():

Parameter	Value	Description
AEEEvent evt	EVT_POINTER_DOWN EVT_POINTER_MOVE EVT_POINTER_UP	Pointer events
uint16 wparam	Unsigned 16-bit value	size of single-byte-character string, including terminating NUL character, pointed to by dwparam, in bytes

Parameter	Value	Description
uint32 dwparam	Pointer to a NUL-terminated single-byte-character string.	dwParam is a pointer to a NUL-terminated single-byte-character string containing pointing device information in the form of delimiter-based “name=value” pairs for x-coordinate, y-coordinate, time, click count, modifiers, ptrID, displayID etc. e.g.: x=00000017,y=FFFFFFFB, time=00003456,clkcnt=2,” where x=+23 and y=-5 in decimal.” x and y are mandatory parameters that need to be set by OEMs. ptrID and displayID are optional parameters that may be generated by OEMs if the device supports only one pointing device or only one display. However ptrID is mandatory if there is more than one pointing devices and displayID is mandatory if there is more than one display. BREW will calculate the time and click count and insert it into the dwParam string before sending the event to the app. See the section below for a detailed description of each parameter.

Generating Pointer Events

OEMs must send pointer events using the AEE_Event() function if their device has a pointing component (stylus, mouse, touch-screen and so on). The event codes can be one of the following:

- AEE_Event (EVT_POINTER_DOWN, wParam, dwParam): This event is sent when the pointing device comes in contact with the screen.
- AEE_Event (EVT_POINTER_MOVE, wParam, dwParam): This event is sent when the pointing device moves on the screen while maintaining contact with the screen.
- AEE_Event (EVT_POINTER_UP, wParam, dwParam): This event is sent when the pointing device loses contact with the screen.

dwParam is a pointer to a NUL-terminated single-byte-character string containing delimiter-based name=value pairs and wParam is the length of the single-byte-character string, including terminating NUL character, pointed to by dwparam, in bytes.

The pointer events can be generated using one of two options:

- If you need to send only the mandatory x and y coordinates for the pointer event and do not have any additional optional parameters, use the helper function AEE_POINTER_SEND_XY(<event code>, x, y) defined in OEMPointerHelpers.h. The helper function will take care of generating the dwparam single-byte-character string in the proper format and use AEE_Event to dispatch the pointer event to BREW.
- If you need to send additional optional parameters along with the mandatory x and y coordinates for the pointer events, use the helper function AEE_POINTER_SET_XY(<pointer to char buffer>, <size of buffer>, x, y) or AEE_POINTER_SET_XY_OFFSET((<pointer to char buffer>, <size of buffer>, offset for x, offset for y) to format the x and y name-value pairs and then append the optional parameters to the single-byte-character string. The latter function uses offsets on top of existing coordinate values. The x-coordinate string should be at AEE_POINTER_XSTR_OFFSET and the y-coordinate string should be at AEE_POINTER_YSTR_OFFSET. After generating the entire single-byte-character string use AEE_Event to send the pointer event to BREW with the parameters as described in the table above. Make sure to NUL-terminate the buffer before sending it using AEE_Event. BREW will duplicate the string and you can deallocate or remove any references to the char string after AEE_Event returns. The size of the single-byte-character string pointed to by dwParam needs to be at least as big as AEE_POINTER_X_SIZE + AEE_POINTER_Y_SIZE + AEE_POINTER_DISPID_SIZE + 1. If your string will be bigger than this size then use CFGI_POINTER_MAX_STR_SIZE to configure the size of the biggest string that may be generated for any of the pointer events. See the section on configuring the maximum string size for pointer events below.

The string composed by OEMs (or auto-generated using AEE_POINTER_SEND_XY helper function) should contain the following name-value pairs separated by the “,” (comma) delimiter:

Name	Value	Description	Helper function(s)	Example
x (Mandatory)	signed integer in decimal	X-coordinate of the pointer relative to the top-left location (0,0) of the display screen. The numeric string should contain 8 characters representing the hexadecimal value of the x-coordinate for the pointer event. . Example: ‘x=00000017,’ for a total lenght of 11 single-byte characters (AEE_POINTER_X_SIZE). The x-coordinate string in the dwParam should be at AEE_POINTER_XSTR_OFFSET.	AEE_POINTER_SEND_XY() Or, AEE_POINTER_SET_XY() Or, AEE_POINTER_SET_XY_OFFSET() Or, AEE_POINTER_SEND_XY_DISPID() Or, AEE_POINTER_SET_XY_DISPID()	“x=00000017,” i.e. x = +23 in decimal
y (Mandatory)	signed integer in decimal	Y-coordinate of the pointer relative to the top-left location (0,0) of the display screen. The string should contain 8 characters representing the hexadecimal value of the sign representing the y-coordinate for the pointer event. Example: “y=FFFFFFFB,” for a total length of 11 single-byte characters (AEE_POINTER_Y_SIZE). The y-coordinate string in the dwParam should be at AEE_POINTER_YSTR_OFFSET.	AEE_POINTER_SEND_XY() Or, AEE_POINTER_SET_XY() Or, AEE_POINTER_SET_XY_OFFSET() Or, AEE_POINTER_SEND_XY_DISPID() Or, AEE_POINTER_SET_XY_DISPID()	“y=FFFFFFFB,” i.e. y = -5 in decimal
ptrID (Optional if only one pointing device is supported, mandatory otherwise)	Unsigned integer in decimal	The ID of the pointing device. The numeric string should contain 2 characters representing the hexadecimal value of the pointing device’s ID. . Example: “ptrID=01,”, for a total length of 9 single-byte characters	-	“ptrID=01,”

Name	Value	Description	Helper function(s)	Example
dispID (optional if only one display screen is supported, mandatory otherwise)	Unsigned integer in decimal representing the AEECLSID of the display screen	The AEECLSID of the display screen on which pointer event occurred. The numeric string should contain 8 characters representing the hexadecimal value of the AEECLSID of the screen on which the event occurred. Example: "dispID=010127d4,", for a total of 16 single-byte characters (AEE_POINTER_DISPID_SIZE).	AEE_POINTER_SEND_XY_DISPID() Or, AEE_POINTER_SET_XY_DISPID()	“dispID=010127D4,” for AEECLSID_DISPLAY1
mousebtn (optional if device doesn’t support mouse)	Unsigned integer in decimal representing the mouse modifier bit-mask.	Mouse-button modifiers to be sent for EVT_POINTER_DOWN to notify BREW which mouse button (defined in AEEPointerHelpers.h) is pressed. The numeric string should contain 8 characters representing the hexadecimal value of the AEECLSID of the screen on which the event occurred. Example: "dispID=010127d4,", for a total of 16 single-byte characters (AEE_POINTER_DISPID_SIZE). Example, if AEE_POINTER_MOUSE_LBUTTON is pressed, “mousebtn=00001,” for a total of 14 single-byte characters.	-	"mousebtn=00001,”

NOTE: The OEMBitmap implementation should take into consideration display rotation in the functions, OEMBitmap_AppToScreen() and OEMBitmap_ScreenToApp(). (The rotation information will be stored in pMe->m_nRotation if the directions from the Display Rotation OEM Note have been followed.) These functions are used by BREW to translate pointer coordinates before they are passed along to apps.

Assume - upright portrait mode is the default display mode.
- x,y were the co-ordinates of the touch
- W, H are the width and height of the touchscreen.

For a 90 clockwise rotation, x, y would have to be adjusted as follows:
xNew = H - y;
yNew = x;

For a 180 clockwise rotation, x, y would have to be adjusted as follows:
xNew = W - x;
yNew = H - y;

For a 270 clockwise rotation, x, y would have to be adjusted as follows:
xNew = y;
yNew = W - x;

Any custom parameters that the OEMs may wish to send along with the pre-defined parameters should be contained in a case-sensitive MIME-style namespace. The name value pair for custom parameters should have the following format: “x-<oem_name>-<param_name>=value”. Example, hypothetically if the BREW Simulator generates a pressure name-value pair, it would append the following string to the dwParam: “x-BREWSimulator-pressure=100,”. Always ensure that the single-byte-character string pointed to by dwParam is NUL-terminated.

For the above example where x = 23 and y = 10 and generating only the mandatory parameters, the single-byte-character string pointed to by dwparam will look like “x=+0000023, y=-0000005,” with a total size of 23 bytes including the terminating NUL-character.

BREW will append the timestamp, clickcount and modifiers to the dwparam and pass it along to the app. OEMs should not generate the following parameters.

Name	Value	Description	Helper function(s)	Example
time (BREW-generated)	unsigned integer in decimal	Time-stamp of pointer event in milliseconds. The unsigned numeric string should contain 8 single-byte hexadecimal characters representing the 32-bit time value as returned by GETUPTIMEMS().This can be used to calculate the time lag between events. Example: “time=003684F9,” for a total length of 14 single-byte characters (AEE_POINTER_TIME_SIZE).	-	“time=00368 4F9,”

clkcnt (BREW-generated)	unsigned integer in decimal	Click count. The unsigned numeric string should contain 1 single-byte hexadecimal character representing the click count. The click count is a running count of the number of pointer clicks received during the OEM-specified duration. For example, if the timeout duration is 500 milliseconds and two EVT_POINTER_DOWN events are received within 500 milliseconds of each other, then AEE_ POINTER_GET_CLICKCOUNT will return 2 for the second EVT_POINTER_DOWN event. This can be typically used to identify double-click events. Example: “clkcnt=2,” for a total length of 9 single-byte characters (AEE_POINTER_CLKCNT_SIZE).	-	“clkcnt=2,”

modifiers (BREW-generated)	A bit-mask represented as a single-byte character string.	Modifiers can be for: - Keyboard keys like Shift, Alt, Control, etc (see EVT_KEY dwParam bits defined in AEEVCodes.h) - Mouse buttons like Middle, Left, Right - Additional pointing device specific flags, described below Keyboard modifiers are represented by 8 single-byte hexadecimal characters corresponding to the 32-bit key modifier state. For example, if KB_RSHIFT, KB_RCTRL, KB_RALT and KB_NUMLOCK are set, the key modifier string will consist of the following substring in the modifier string: "0004002A". Mouse modifiers are represented in the same way as the keyboard modifiers and are appended to the keyboard modifier substring to generate the "modifiers" string. The only difference is that the mouse modifiers are 16-bit values and are represented by 4 single-byte hexadecimal characters in the modifier string. For example, if AEE_POINTER_MOUSE_LBUTTON is pressed, the mouse modifier substring will be "0001". The additional pointing device modifier flags will provide additional information specific to the pointer events. These are 16-bit values represented by 4 single-byte hexadecimal characters in the modifier string. For example, if a pointer move event is dropped, the AEE_POINTER_EVENT_DROPPED flag is set and the additional modifiers string will be "0001". Example: "modifiers=0004002a00010001," for a total length of 27 single-byte characters (AEE_POINTER_MODIFIERS_SIZE). The modifiers string in the dwParam will be at AEE_POINTER_MODIFIERS_OFFSET. Use AEE_POINTER_GET_KEY_MODIFIERS() or AEE_POINTER_GET_MOUSE_MODIFIERS() to obtain the values. AEE_POINTER_IS_EVENT_DROPPED() will return TRUE if a move event was dropped and FALSE otherwise.	-	“modifiers=0004002A00010001,”

Pointer events use-case scenarios

The following table provides some example use-case scenarios.

Scenario	OEM-generated AEE_Event()	Event delivered to BREW app
Finger touches the primary screen of single-display touch-screen device	eCode = EVT_POINTER_DOWN wParam = 23 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,”	eCode = EVT_POINTER_DOWN wParam = 73 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,time=00367ECE, clkcnt=1,modifiers=0000000000000000,” where x is 54 in decimal and y is 148 in decimal.
Finger is lifted from primary screen of single-display touch-screen device	eCode = EVT_POINTER_UP wParam = 23 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,”	eCode = EVT_POINTER_UP wParam = 73 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,time=00367F0D, clkcnt=1,modifiers=0000000000000000,” where x is 54 in decimal and y is 148 in decimal.
Finger touches down again within 500ms (default multi-click timeout) of first touch on primary screen of single-display touch-screen device	eCode = EVT_POINTER_DOWN wParam = 23 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,”	eCode = EVT_POINTER_DOWN wParam = 73 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,time=00367F1D, clkcnt=2,modifiers=0000000000000000,” where x is 54 in decimal and y is 148 in decimal.
Finger moves across primary screen of single-display touch-screen device after double-click	eCode = EVT_POINTER_MOVE wParam = 23 (including NUL-terminating character) dwParam = “x=0000002F,y=00000098,”	eCode = EVT_POINTER_MOVE wParam = 73 (including NUL-terminating character) dwParam = “x=0000002F,y=00000098,time=00367F2C, clkcnt=2,modifiers=0000000000000000,” where x is 47 in decimal and y is 152 in decimal.
Stylus touches down on secondary screen of multi-display touch-screen device with KB_RSHIFT, KB_RCTRL, KB_RALT, and KB_CAPSLOCK pressed on the keyboard	eCode = EVT_POINTER_DOWN wParam = 39 (including NUL-terminating character) dwParam = “x=0000003A,y=00000008, dispID=010127D4,”	eCode = EVT_POINTER_DOWN wParam = 89 (including NUL-terminating character) dwParam = “x=0000003A,y=00000008, time=00446DD3,clkcnt=1, modifiers=0001002A00000000, dispID=010127D4,” where x is 58 in decimal and y is 8 in decimal and displayID is AEECLSID_DISPLAY1.

Left-button on mouse is pressed on primary screen of multi-display device with KB_RSHIFT and KB_RCTRL pressed on the keyboard	eCode = EVT_POINTER_DOWN wParam = 53 (including NUL-terminating character) dwParam = “x=00000007,y=0000001C, dispID=010127D4, mousebtn=0001,”	eCode = EVT_POINTER_DOWN wParam = 89 (including NUL-terminating character) dwParam = “x=00000007,y=0000001C, time=00509A47,clkcnt=1, modifiers=0000000A00010000, dispID=010127D4,” where x is 7 in decimal and y is 28 in decimal and displayID is AEECLSID_DISPLAY1 and mouse button is AEE_POINTER_MOUSE_LBUTTON.
Mouse is moved with left-button pressed on primary screen of multi-display device with no keys pressed on the keyboard	eCode = EVT_POINTER_MOVE wParam = 53 (including NUL-terminating character) dwParam = “x=00000007,y=0000001C, dispID=010127D4, mousebtn=0001,”	eCode = EVT_POINTER_MOVE wParam = 89 (including NUL-terminating character) dwParam = “x=00000007,y=0000001C, time=00509A47,clkcnt=1, modifiers=0000000000010000, dispID=010127D4,” where x is 7 in decimal and y is 28 in decimal and displayID is AEECLSID_DISPLAY1 and mouse button is AEE_POINTER_MOUSE_LBUTTON.
Stylus moves across primary screen of single-display touch-screen device very rapidly resulting in stale move events in the event queue	eCode = EVT_POINTER_MOVE (multiple such events) wParam = 23 (including NUL-terminating character) dwParam = “x=00000036,y=00000094,” where x is 54 in decimal and y is 148 in decimal.	eCode = EVT_POINTER_MOVE wParam = 73 (including NUL-terminating character) dwParam = x=00000036,y=00000094, time=00367F0D,clkcnt=1, modifiers=0000000000000000,” Followed by some EVT_POINTER_STALE_MOVE events depending on how many move events got queued. eCode = EVT_POINTER_STALE_MOVE wParam = 73 (including NUL-terminating character) dwParam = x=0000004A,y=00000075, time=00368001,clkcnt=1, modifiers=0000000000000000,” . . . Followed by EVT_POINTER_MOVE when the event queue doesn’t have any more pending move events: eCode = EVT_POINTER_MOVE wParam = 73 (including NUL-terminating character) dwParam = x=00000054,y=0000005E, time=00377AB0,clkcnt=1, modifiers=0000000000000000,”

Backward compatibility for pen events

To maintain backward compatibility, BREW will still generate EVT_PEN_* events if the application doesn't handle EVT_POINTER_* events. EVT_PEN_* events are, however, considered deprecated and newer applications should handle EVT_POINTER_* events instead of EVT_PEN_* events.

Additional configuration for pointer events

Configuring click count time-out for pointer events

CFGI_POINTER_MULTICLICK_TIME is the number of milliseconds during which pointer clicks will be treated as multi-clicks (e.g. double-click) by BREW. Default is AEE_POINTER_MULTICLICK_TIME (500ms).

You may control this behavior by modifying the value of click timeout as follows:

```
case CFGI_POINTER_MULTICLICK_TIME:
{
   uint32 * pclktime = (uint32 *)pBuff;

   if (!pBuff || nSize != sizeof(uint32))
      return(EBADPARM);

   if (pclktime)
   {
      *pclktime = NNNN;// click time-out in milliseconds
      return SUCCESS;
   }
}
```

Please refer to documentation in sdk/inc/AEEPointerHelpers.h and pk/inc/OEMPointerHelpers.h for more information on various helper functions and definitions.

Sending pen events

Starting with BREW Porting Kit (MSM) 3.1.4SP02, pen events (EVT_PEN_DOWN, EVT_PEN_MOVE, EVT_PEN_UP and EVT_PEN_STALE_MOVE) will be marked as deprecated. Instead, Pointer events should be used. See Sending pointer events on page 44 for details.

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. The events and their parameters appear in the tables below.

AEE_Event joystick events sent by OEMs

AEE_Event() (EVT_JOYSTICK_POS)	Sent when the joystick moves.

AEE_Event joystick event sent by BREW

AEE_Event() (EVT_JOYSTICK_STALE_POS)	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-fourths 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 **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	Upper 16 bits = the signed x-coordinate
	The joystick position of the y coordinate on the display	Lower 16 bits = the signed y-coordinate

Implement essential OEM Layer

This section discusses the implementation of the most essential OEM Layer code, which must be implemented on any device on which BREW will run.

Implementing Operating System support

BREW uses services of the device's operating system for functions such as getting the current time, setting timers, locking and unlocking interrupts and signaling the task in which BREW is running. OEMs must implement the functions defined in OEMOS.h to start BREW. See the *OEM API Reference Online Help* for more details on the OEMOS APIs, which must be implemented.

Implementing File System support

BREW uses the device file system as the storage for BREW applications, preferences, and application content. BREW also makes the device file system services available to BREW applications for their storage needs.

To implement File System support for BREW, see BREW Services: File System on page 82.

Configure the Device

This section describes the configuration items and tells you how to configure BREW Heap, MCF, and RUIM-based devices. Before BREW can operate properly, you must implement the configuration functions listed in this section.

Configure config items

The following functions are used to retrieve and set the configuration parameters including the ones for download services (CFGI_DOWNLOAD):

- OEM_GetConfig()
- OEM_SetConfig

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

AEEConfigItem is the data type passed to these two functions. OEMs must implement support for all the defined config items, which are described in the BREW OEM API Reference Online Help under the OEM_GetConfig and OEM_SetConfig functions. In particular, CFGI_DOWNLOAD related parameters must be settable on the devices that will support BREW download.

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 implementations to verify that they are suitable for their device.

Configure device items

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

- OEM_GetDeviceInfo()

- OEM_GetDeviceInfoEx()

AEEDeviceInfo is the data type passed to OEM_GetDeviceInfo(), and 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 Online Help* under the AEEDeviceInfo and AEEDeviceItem data types. The items defined are in the file, AEEShell.h.

Language setting is the dwLang field of AEEDeviceInfo structure. BREW queries this field to obtain the current language setting. When OEMs change the language either in the native or BREW state, this dwLang field must be updated accordingly when it is queried. In addition to that, OEMs must also call the function, AEE_IssueSystemCallback(AEE_SCB_DEVICE_INFO_CHANGED) to trigger BREW flushing cached resources. Any application that registers this system callback through ISHELL_RegisterSystemCallback() will also be notified by the callback object so that the applicantion can reload the text strings in the new language.

Configure BREW heap

The BREW heap is used by all BREW applications. The size and number of BREW applications, which can execute simultaneously on a device, are dictated by the amount of heap made available to BREW. See BREW Services: Heap on page 93 for more information and Compile OEM source files on page 69 and Appendix C: Test Enable Bit Removal on page 210 for debugging information.

Configure MCF

The MCF (Media Content File) provides a method to store media files in designated directories, which can only be accessed by authorized application. See the *BREW API Reference Online Help* for a description of the Multimedia_Content_File.

To port the MCF feature, OEMs should include all the OEMMCFMIF_xxxxx.c files under pk\src in the Makefile. Then pk\src\OEMConstFiles.c should be modified by adding all the gxxxxx_MIF variables defined by OEMMCFMIF_xxxxx.c in the gpOEMConstFiles[] array.

The directories for these MCF categories will be predefined on a device. The mapping between MCF macros defined in AEEMCF.h and the corresponding directories in the File System can be found in the *BREW API Reference Online Help*. To verify the porting process, run OATFS of PEK 3.1.5. See the *BREW Porting Evaluation Kit User Guide*.

NOTE: You don't need to create the MCF directories ahead of time. On first write attempt, these directories will be automatically created for you.

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. See Implementing file system access restrictions for diagnostic tools on page 88.

Configure IPosDet functions

To correctly port the IPosDet interface, OEMs must configure information for the following two functions in OEMPosDet.c.

- OEM_PD_SharedVoiceReceiver()
- AEEGPSInfo.LocProbability

OEM_PD_SharedVoiceReceiver()

Configure the OEM_PD_SharedVoiceReceiver() function as follows depending on the receiver configuration:

- If the device has a shared receiver for GPS and voice calls and it is unacceptable to have tune-away during a voice call for a position determination request, return TRUE from the function OEM_PD_SharedVoiceReceiver() in OEMPosDet.c.
- If the device has a shared receiver for GPS and voice calls and tune-away during a voice call for a position determination request is acceptable, return FALSE from the function OEM_PD_SharedVoiceReceiver() inOEMPosDet.c.
- For devices which support simultaneous voice and GPS (do not have a shared receiver for GPS and voice calls), return FALSE from the function OEM_PD_SharedVoiceReciever() in OEMPosDet.c

AEEGPSInfo.LocProbability

The position response returned by the IPosDet interface is the AEEGPSInfo structure. From BREW 3.1.3 onwards, this structure contains a field which indicates the probability that the user's position is within the ellipse described in the response. This field is AEEGPSInfo.LocProbability.

As part of porting the IPosDet interface, OEMs must verify the probability specified in OEMPosDet.c in function OEM_PD_EventCB() is accurate for the device.

The position determination engine on some devices provides the probability along with the position response. For these devices, the value provided by the position determination engine is the information that should be returned from the OEM layer to the application.

Example:

```
//The position determination engine provides the information.
pMe->theResponse.LocProbability = pd_info_ptr-
>pd_info.pd_data.loc_uncertainty_conf;
```

For devices where the position determination engine does not provide this information directly, the OEM can configure the value returned for the location probability.

The location probability is determined by the position determination engine on the device. For 1x MSM chipsets, the location probability is typically 39% or 68%. The correct value to use must be verified based on the position determination engine in that device.

Example:

```
//The OEM configures the location probability after verifying the
//correct value for that chipset:
pMe->theResponse.LocProbability = 68;
```

Integrate the native UI in BREW enabled devices

This section tells you how to integrate the native UI in BREW enabled devices. The following process describes the steps you must perform to support the coexistence of BREW and non-BREW applications, also referred to as native applications, on a device. The primary goal of the process is to enable a user to switch from one application to the other, regardless of whether the application is a BREW or native application. For instance, a user can switch from a game (BREW-based) to a browser (native application) and return to the same instance of the same game when finished using the browser.

Choose native UI interaction with BREW

Shim vs. suspend/resume

The suspend/resume approach allows for BREW and native applications to maintain and manage their own application list. Maintaining two separate application lists can increase the complexity of state management on your system.

The shim approach allows for only one application list to be maintained on the device by BREW; this list is common to both BREW and native applications. This helps BREW maintain and adjust the application list including the native applications, and run each of the native applications in a BREW application context. In addition, a single application stack of BREW applications can mix with native applications.

Implement Suspend/Resume

Suspend and resume BREW after initialization and before termination through AEE_Suspend() and AEE_Resume(). Typically, you suspend BREW when a task on your device needs to take control of the UI, for example, when a non-BREW SMS or a call arrives, or a status alert (low battery warning) needs to be displayed.

AEE_Suspend() must be invoked only when there is a need for the native software to draw to the screen or take control of the keypad. If the native software must display the message on the screen, call AEE_Suspend() prior to displaying the message. When the message is removed from the screen, call AEE_Resume() to resume BREW.

The following is the sequence of steps to suspend BREW and launch a non-BREW application. A typical example occurs when the device receives an incoming call while a BREW application is running. When the call is received, AEE_Suspend() is called to launch the native UI call dialog. When the call is completed, AEE_Resume() is called to resume the BREW application.

To suspend BREW and launch a non-Brew application

1. Launch a BREW application while in BREW.
2. Call AEE_Suspend() to suspend BREW when the device receives an incoming call.

   BREW sends the EVT_APP_SUSPEND event to suspend the current BREW application, and the native UI displays the call dialog (or any non-BREW application).
3. Close the call dialog (native application) started in step 2, when the call ends.
4. Call AEE_Resume() to resume the BREW application.

   BREW sends the EVT_APP_RESUME event to the suspended BREW application and allows the application to resume.

See the *OEM API Reference Online* Help for more information on the following topics:

- AEE_Suspend()
- AEE_Resume()

Implement BREW shim

Whenever the native software layer takes control of the display, a corresponding BREW shim application is started. The purpose of this shim application is to ensure that drawing to the screen is done in the context of a BREW application. For all BREW purposes, a BREW application is running. BREW maintains an application list, which is 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. When you switch between native and BREW applications, this could cause issues, which can be avoided using the shim approach.

The following is a sequence of steps in which a BREW shim application is started when the native layer wants to take control of the display to display an incoming call dialog. Once the call ends, the BREW shim application is closed, and the BREW application that was suspended earlier is resumed automatically by BREW.

1. The device receives an incoming call as a BREW application is running.
2. A BREW shim application starts by calling ISHELL_StartApplet().
3. The BREW application is automatically suspended.
4. The BREW shim application, advising that the user is in the call, puts up a UI dialog corresponding to the call.
5. The call ends.
6. The BREW shim application closes by calling ISHELL_CloseApplet().
7. BREW automatically resumes the application that was running when the call was received.

See *BREW UI: Integrating native apps with BREW* on page 180 for more information on the following topics:

- Shim applications

Enable BREW Application Manager

This section describes the BREW Application Manager and points you to procedures for integrating it.

The BREW Application Manager provides the user interface that allows users to browse applications installed on a device, launch them, or remove them. The BREW Application Manager package also includes Mobileshop®, which is the user interface into the carrier's application catalog. Mobileshop allows the user to browse the catalog and purchase or upgrade BREW Applications. BREW Application Manager is an important part of the BREW solution and must be incorporated into your device in accordance with the Carrier's requirements. See the *BREW Application Manager Guide* and the Mobileshop documentation for more information.

Select BREW Application Manager version

OEMs should check the carrier's requirements or any other applicable agreement with their carrier customer to confirm that the correct version of the BREW Application Manager is incorporated into their device.

Integrate BREW Application Manager

Once you have determined the proper version of BREW Application Manager, QUALCOMM recommends that you incorporate BAM into your device build by adding the application and its resources to BREW's "Persistent File System". This includes the .mod, .mif, .bar files for each application you wish to incorporate into your build. See Methods of enabling features on page 70 for more information on building applications with your device image.

Please refer to the documentation in the BREW Application Manager package for a description of the BAM applications and more specific information on how to integrate BREW Application Manager with your device build. Pay particular attention to any prerequisites that are required by the version of BAM you select. In particular BAM 3.x has been written on top of BREW UI Widgets (BUIW). Therefore you must also incorporate BUIW into your build if you wish to use BAM 3.x.

NOTE: While integrating the BREW Application Manager, be sure to refer to your Carrier's requirements. Some carriers require customizations on top of the baseline BREW Application Manager that QUALCOMM provides. Some of these customizations may be achieved simply by modifying some compile-time parameters, while others may require more work. Also, be aware of which languages your device must support and be sure to integrate the resource (BAR) files for each language that must be supported.

Perform the first device build

At this point you are ready to generate your first BREW-enabled device build. This section describes how to integrate the BREW sources and libraries into your build. This step may vary depending on your particular build system, but will consist of the general steps explained below.

Adding persistent files

See Creating persistent files on page 86

Compile OEM source files

All the OEM Files you have implemented as part of porting BREW must be compiled. You may have to modify your device build's makefile(s) to add all of the OEM sources to the list of files that need to be compiled.

Although the BREW OEM Porting Kit contains some AEE source files, these files are not intended to be rebuilt by OEMs. They are already pre-built into the BREW libraries and are provided solely to aid OEMs in debugging efforts. They must not be modified.

Link BREW libraries

The BREW OEM Porting Kit comes with several sets of libraries. The libraries are installed to different directories with descriptive names. For example, the ADS12arm7 directory contains a set of BREW libraries that were built with ARM Developer Suite (ADS) 1.2 for the ARM7 CPU. Within these directories, you may find a directory named AEEPrefixLibs. This directory contains a subset of the BREW libraries where all the symbols are prepended with the prefix AEE. You should only use these prefix libraries if you run into linker errors due to symbol clashes with other pieces of your device software. Once you have identified the set of libraries that are appropriate for your device, you need to add the libraries to the list of libs that are linked together to produce your final device build.

Enable BREW Features

This section explains how to enable the maximum set of BREW features your device will support. See the BREW Services sections for more information on enabling specific BREW features. BREW is flexible in that it allows OEMs to enable or disable many of the core APIs, which are published in the BREW SDK. QUALCOMM and many operator specifications recommend that OEMs enable as many of the core BREW APIs as their device can technically support. BREW also allows OEMs to develop their own custom APIs, even BREW applications, and include them in their device build.

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 include 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).

Methods of enabling features

BREW supports several methods of enabling or adding features to device builds. Each method has its own advantages:

- Statically linked classes or modules (via OEMModTable.c and OEMModTableExt.c). Most of the core BREW APIs are linked into the device build in this way. OEMModTable.c and OEMModTableExt.c are tables containing a list of function pointers that point to entry points for the static modules or classes. With this method, the modules are part of the device build and may not be removed from the device by the user.

  NOTE: Beginning in BREW 3.1.0, statically-linked modules MUST have a MIF file and this MIF file MUST be part of the Persistent File System.

- Pre-loaded modules. Pre-loaded modules are dynamic modules that are loaded into the device's file system during manufacturing. This method must be coordinated with QUALCOMM or the operator since pre-loaded modules must be signed. See BREW Services: Pre-loading applications on page 99 for more information about this method.

- Persistent File System. See Creating persistent files on page 86.

Statically linked modules and classes

Modules and classes are statically linked to BREW via OEMModTable.c and OEMModTableExt.c. OEMModTable.c is intended to be used unmodified. This file contains entry points to many of the core BREW APIs. You may control the compilation of this file by defining or undefining FEATUREs in OEMFeatures.h.

Non-core modules and classes may be statically linked with the device build by adding the entry points to OEMModTableExt.c. This file contains two key structures that should be modified for adding statically linked modules or classes: gOEMStaticModList and gOEMStaticClassList.

Static Module List

gOEMStaticModList is a list of AEEStaticMod structures. AEEStaticMod contains the following:

- char* pszMIF. NULL-terminated string corresponding to the absolute BREW path to this module's MIF file, for example, fs:/mif/brewappmgr.mif.
- PFNMODENTRY pfnModEntry. Pointer to the Module _Load function. This function returns an IMODULE Interface to the caller.

In AEEStaticMod you should add any applications that you want to link into your build as static modules. Your module contains a _Load() function of the following signature:

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

The purpose of this function is to create an IMODULE interface and return it to the caller. You must also compile and link this module with your device build. A sample implementation of a _Load() function can be found in xmodstub.c and xmodimpl.c.

Static Class List

gOEMStaticClassList is a list of AEEStaticClass structures. The AEEStaticClass structure contains the following fields:

- AEECLSID cls. The Class ID of the class this structure refers to.
- uint16 wFlags. any combination of the following:

  - ASCF_UPGRADE. If set, a downloaded implementation of this Class ID will take precedence over the statically-linked implementation.
  - ASCF_PRIV: If set, the Class ID for this class must be listed in the External Classes section of the MIF file of any application that wishes to call ISHELL_CreateInstance() for this class.
- uint16 wMinPriv. The Legacy Privilege level applications must have to call ISHELL_CreateInstance() for this class. This field is largely deprecated in favor of setting the ASCF_PRIV bit in the wFlags member.
- void* pfnInit: Initialization function to be called during AEE_Init(). If the class needs to execute any initialization code when BREW initializes, set pfnInit to point to the initialization function. If no initialization is required, set this field to NULL.
- int* pfnNew: New function to be called to create a new instance of the class. When an application calls ISHELL_CreateInstance() for this Class ID, this function will be called to actually create the interface and return a pointer to it.

If the feature you wish to add is a Static Extension that must run initialization code when BREW initializes, you should add that class to the gOEMStaticClassList and specify an Init function.

The Init() function must have the following signature:

```
void MyExtensionClass_Init(IShell * ps);
```

The purpose of the Init() function is to perform any necessary power-up initialization that may be required by your extension class.

The New() function must have the following signature:

```
int MyExtensionClass_New(IShell * ps, AEECLSID cls, void ** ppobj);
```

The purpose of the New() function is to MALLOC storage for an instance of the class specified in the cls parameter and return a pointer to the new instance of the caller via the ppobj.

Example 1

This example shows how to add the HelloWorld module as a statically-linked module.

The HelloWorld module must have a _Load() function. You need to extern this function near the top of OEMModTableExt.c as follows:

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

Then add a line to the gOEMStaticModList as follows:

```
static const AEEStaticMod gOEMStaticModList[] =
{
   {AEEFS_MIF_DIR"helloworld.mif", HelloWorld_Load},
   {NULL, NULL}
};
```

Finally add the Helloworld sources to the device build so that they are compiled when the device is built.

Example 2

Following is an example showing how to add the IMYEXTCLS extension to the static class table with the assumption that this extension needs to run an initialization routine on device power-up.

The IMYEXTCLS extension contains both an Init() and a New() function. You need to extern these functions near the top of OEMModTableExt.c as follows:

```
extern void IMYEXTCLS_Init(IShell * ps);
extern int  IMYEXTCLS_New(IShell *ps, AEECLSID cls,void ** ppobj);
```

Next add an entry to gOEMStaticClassList as follows:

```
static const AEEStaticClass gOEMStaticClassList[] = {
   {AEECLSID_MYEXTCLS, ASCF_UPGRADE | ASCF_PRIV, 0, IMYEXTCLS_Init,
IMYEXTCLS_New},
   {0,0,0,NULL,NULL}
};
```

Pre-Loaded Modules

A Pre-loaded module is a dynamic BREW module that is loaded into the device's file system during manufacturing. These modules must be signed by Qualcomm or the operator, so you need to coordinate with both. See BREW Services: Pre-loading applications on page 99 for details on incorporating pre-loaded modules into your device.

Run the PEK

This section describes the Porting Evaluation Kit (PEK) and tells you where to find it and how to use it to test your porting process.

The PEK consists of tools and documentation to verify/test BREW porting to a device. The PEK tools also measure the device performance for the most commonly used BREW services, such as display and file system.

PEK location

The latest version of the PEK is available on the BREW OEM Extranet as well as an archive of previous PEK releases. Make sure you download the correct version as well as any patches released in correspondence with the BREW version on your device. The version is usually determined by your carrier. Refer to your carrier's requirements when determining the PEK version that corresponds to your BREW version.

Using the PEK

When you download and install the PEK, you receive the following:

- PEK Studio, a tool that performs interface testing for certain BREW tools.
- OAT modules with source code, the test cases for the BREW interfaces that test/verify the functioning of the BREW APIs.
- BREWStone®, a module that measures the performance of a device.
- PEK documentation, which includes BREW PEK Test Cases and the BREW PEK User's Guide

If you are new to PEK, you should read the *BREW PEK User Guide* to set up and use the PEK Studio to conduct the tests.

In addition to PEK, you need to install the BREW Device Configurator and read the BREW Device Configurator documentation to create a Device Pack. A Device Pack is a special file format that holds all relevant information about your device so that the PEK test cases can test/verify the port.

PEK should be run as often as possible during the porting process. You will notice that there is an OAT module for each BREW interface. When porting for a particular interface or interfaces is done, at least the OAT module corresponding to those interfaces should be run. This will ensure that porting discrepancies are exposed as soon as possible.

After the porting process is complete, PEK should be run anytime there is a BREW or non BREW change made to the device build sources. This will prevent any errors due to the changes made.

When you encounter OAT test failures, the failed test results and the *PEK Test Case Online Help* will provide clues to the source of the problem. PEK also comes with source code for the OAT modules. Refer to the source code to understand more about the test cases. See BREW OEM Support on page 10 for help on PEK test failure issues.

BREW Services: R-UIM and SIM support

This section provides the set of procedures to be followed by OEMs when implementing the R-UIM feature on an R-UIM based BREW device.

R-UIM interface

The R-UIM interface is a collection of functions that do the following:

- Verify the R-UIM card connection
- Return the current R-UIM status
- Compare the designated Card Holder Verification (CHV) on the R-UIM with the PIN passed
- Change the designated CHV on the R-UIM to the PIN passed.

Overview of an R-UIM based device

In non-R-UIM based systems, the ESN is used for authentication of mobile stations through the Cellular Authentication and Voice Encryption (CAVE) algorithm. R-UIM based mobile stations can use either the Mobile Equipment (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.

While some carriers require that UIMID be used for authentication 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, or through OEM_GetConfigEx() for the item AEE_DEVICEITEM_HWIDLIST.

BREW on an 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 HWID, are used while encrypting the files. BREW allows only valid and properly encrypted application devices. When applications are moved to another device or if the characteristics of the device, such as the HWID, 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 primary HWID 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 an R-UIM based device:

To port Brew on an 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 to access AddrBook on an R-UIM device.

   To create an instance of IAddrBook to access the address book on an R-UIM device, the ClassID, AEECLSID_ADDRBOOK_RUIM, is 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. After BREW is initialized by calling AEE_Init(), if and when the SID changes, BREW must be notified of the new value via the IDOWNLOAD_SetSubscriberID() method.

It is possible that R-UIM initialization is not completed when AEE_Init() returns. The same procedure of resetting the SID should apply after R-UIM is fully initialized. It is critical that BREW is informed of the changed SID because this information is used for billing purposes. The following code must be invoked to notify BREW of the SID change:

```
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;
}
```

4. BREW configuration parameters obtained through OEM_GetConfig() must come from either the ME or the R-UIM. Most items are from the ME. Only the following items or sub-items should be from the R-UIM card:

   CFGI_SUBSCRIBERID
   CFGI_SUBSCRIBERID_LEN
   CFGI_CARDID
   CFGI_CARDID_LEN
   CFGI_GSM1X_IDENTITY_PARMS
   CFGI_GSM1X_PRL
   CFGI_GSM1X_RTRE_CONFIG
   CFGI_GSM1X_SID_NID_PARAMS
   nCurrNAM of AEEMobileInfo in CFGI_MOBILEINFO
   szMobileID of AEEMobileInfo in CFGI_MOBILEINFO
   bAKey of AEEDownloadInfo in CFGI_DOWNLOAD

   See OEM_GetConfig and OEM_SetConfig in the *OEM API Reference Online Help*.

5. 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

6. Ensure that a valid ME ESN is returned inside OEM_GetConfig() for CFGI_MOBILE_INFO and a valid HWID is returned for OEM_GetConfigEx() for AEE_DEVICEITEM_HWID. The following are the HWID requirements:

   - The dwESN member of the AEEMobileInfo structure must be a valid ME ESN and must uniquely identify this device. If the device does not have an ME ESN assigned, a pESN should be used.
   - The HWID must be kept constant and must not change when the R-UIM card is swapped.
   - The HWID is also used by application developers while generating test signatures for the device, so you must allow application developers to obtain the HWID of a device.

7. 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 may 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.

8. 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.

9. 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.

10. Provide a menu item which shows the mobile station's HWID. This item is critical to developers who need to create test signatures based on the HWID.
11. 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. The recommendation is to use a pESN.

    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 an R-UIM based device.

To run test 1

1. Download subscription and non-subscription BREW applications on an 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 it is 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 it is not set, you can run only non-subscription applications.

To run test 2

1. Download BREW applications on an 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

1. 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.

BREW Services: File System

This section lists BREW File System services, how to implement the BREW FIle System, and concrete BREW example paths and their meanings.

File System services

BREW requires an internal/non-removable file system for it to function. The file system should provide the following services to BREW:

- Creating new files and directories
- Removing existing files and directories
- Truncating existing files
- Seek and Tell operations on open files
- Enumeration of existing files and directories
- Return information such as file attributes and available free space
- A tool/mechanism to access this file system externally and perform tasks that include, but are not limited to the following:
  - Enumerating files and directories
  - Providing the total size of EFS and the available size at a particular instance
  - Navigating through the directory structure
  - Creating files and directories
  - Removing existing files and directories

For the MSM family of chipsets, BREW accesses the EFS. EFS provides all of the above listed services to BREW. Also, QCT provides a tool within the QCT Test Suite called the EFS Explorer that provides all of the above functionalities and more.

The BREW Tools Suite also contains a similar tool called the BREW AppLoader, which will perform some of the same services listed under the file system services.

Implementing the BREW File System

BREW exposes the IFileMgr and the IFile set of API's for BREW applications to access the BREW File System (both the internal file system and the external/removable memory devices) as defined in OEMFS.h. To implement these file system API's, OEM's need to implement the functions. Reference implementations for these functions are provided in OEMFS.c. For more information, see the *BREW OEM API Reference Online Help*.

Since BREW 3.0, the canonical BREW file namespace begins with the string "fs:/" and allows addressing of the entire BREW file system. For example, 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. Qualcomm recommends to map to the /brew/ directory
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/.	The modules directory, currently considered the application directory in the SDK.
fs:/mod/	
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 does not 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:/~/.

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/

OEMs are responsible for mapping BREW namespace names in OEMFSPath.c and OEMFS.c to their proper locations in their native file system.

When mapping the BREW name space names to the native file system, the following should be considered:

- 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.

Previously, dynamic applications were restricted to naming (and therefore accessing) only those files and directories under their module directory (with the exception of the Shared directory), ringers directory, and possibly the addrbk directory.

Since BREW 3.0, dynamic applications can name anything, but access restrictions apply. First, a module's permissions must include a PL_FILE to be able to modify the BREW File System. Modules may publish or export files in their home directories. Also, BREW looks for the RESTYPE_BINARY resource IDB_MIF_ACLS (ACLS in the MIF Editor) in each module's MIF to resolve which files are published with what permissions. In this resource, BREW expects to find a doubly-NULL-terminated string with the following syntax:

```
ACLs= (ACL '\0')+ '\0'
ACL            = assignperms ":" path
path          = <single-byte string>
assignperms    = assignperm [ ";" assignperms ]
assignperm     = groups "=" exactperms "/" subtreeperms
groups         = group [ "," group ]
group          = <hex number>
exactperms     = perms
subtreeperms   = perms
perms         = [+-] [r] [w]
```

For more information about creating ACLs in the MIF Editor, see the *BREW Programming Concepts* that comes with the SDK and the Managing ACLs document in the *BREW SDK Tools User Docs*.

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.

The steps involved in adding the BREW module, HelloWorld, to your device build illustrates these capabilities.

The HelloWorld module contains the following:

```
helloworld.mif – BREW Module Information File
helloworld.mod – Binary module
helloworld.bar – BREW Resource File
```

BREW Services: File System on page 82 illustrates that these files will need to be loaded onto the device in the following locations:

```
fs:/mif/helloworld.mif
fs:/mod/helloworld/helloworld.mod
fs:/mod/helloworld/helloworld.bar
```

To add HelloWorld to your device

1. Run the BIN2SRC utility to convert these binary files into source files that can be compiled as part of the device build. Execute the following command lines to convert these files into C source files.

   ```
   bin2src –shelloworld.mif -dfs:/mif/helloworld.mif -
   ohelloworldmif.c
   bin2src –shelloworld.mod -dfs:/mod/helloworld/helloworld.mod -
   ohelloworldmod.c
   bin2src –shelloworld.bar -dfs:/mod/helloworld/helloworld.bar -
   ohelloworldbar.c
   ```

   Three source files are produced: helloworldmif.c, helloworldmod.c and helloworldbar.c. Inside each of these files you will find code that looks like this:

   ```
   const AEEConstFile gHELLOWORLD_MIF =
   "fs:/mif/helloworld.mif",FALSE,4576,0,4576, (byte*)&gsData};
   const AEEConstFile gHELLOWORLD_MOD =
   "fs:/mod/helloworld/helloworld.mod",FALSE,14389,0,14389,
   (byte*)&gsData};
   ```

```
const AEEConstFile gHELLOWORLD_BAR =
"fs:/mod/helloworld/helloworld.bar",FALSE,31523,0,31523,
(byte*)&gsData};
```

2. Add each of these symbols to the OEMConstFiles.c.
3. At the top of OEMConstFiles.c add extern declarations:

```
extern AEEConstFile gHELLOWORLD_MIF;
extern AEEConstFile gHELLOWORLD_MOD;
extern AEEConstFile gHELLOWORLD_BAR;
```

4. Modify the declaration of gpOEMConstFiles by adding the lines shown in red:

```
static const AEEConstFile *  gpOEMConstFiles[] = {
                        gHELLOWORLD_MIF,
                        gHELLOWORLD_MOD,
                        gHELLOWORLD_BAR,
```

5. Add helloworldmif.c, helloworldmod.c and helloworldbar.c to your device build.

Implementing file system restrictions

BREW and BREW applications make extensive use of the file system to store applications, resource files, databases, preferences, settings, cache files, and so forth.

The basic parameters to control usage of the file system are the following:

- Number of files
- Number of open files
- Maximum file name and file path length
- Special character support in file names

Depending on the file system size, the values and functions controlling the above parameters must be adjusted. 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()

Details on how to configure the EFS for a given flash size and available RAM can be found in the System Parameters chapter of the *Embedded File System Interface Specification and Operational Description (ISOD)*.

Implementing file system access restrictions for diagnostic tools

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 or USB interface using diagnostics commands. OEMs are required to implement the “BREW file access restrictions” to help ensure that BREW applications and their content are protected from unauthorized copying.

To implement BREW file access restrictions, the following steps are required:

1. The registration function OEMFS_RegRmtAccessChk() must be implemented. BREW calls this registration function during AEE_Init(). The purpose of this function is to register a callback function that must be called each time a file access is attempted remotely.
2. The OEM Layer code must call the callback function that was registered in Step 1 each time a file access is attempted remotely. When this callback is invoked, BREW will apply the following set of rules in the order listed to decide whether to allow or deny access to the file specified:
   - Listing files is always allowed.
   - Any access to the fs:/ringers directory will be denied.
   - Any access to the fs:/download directory will be denied.
   - Any access to the fs:/sys/priv directory will be denied.
   - For numerically-named directories (for example, fs:/1234):
     - Files may not be read
     - Files may not be deleted
   - Any access to a subdirectory named “private” of a numerical-named directory will be denied.
   - For numerically-named files with .mif extension in the fs:/mif directory
     - Files may not be read.
     - Files may not be deleted.

Based on the response of the callback function, the OEM Layer code must allow or deny access to the files where access was attempted.

OEMFS_RegRmtAccessChk() 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 AMSS files provide this implementation.

Porting instructions

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

This feature requires a patch to the AMSS. The access restriction and memory operations features are not included in all AMSS 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 AMSS definitions described above indicates that the build contains these features. If you find that the build does not include these features, obtain the AMSS patch by contacting brew-oem-support.

Step 2. Enable the feature in the device build.

After obtaining the AMSS 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 91.

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 AMSS changes even if the version of AMSS 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);
```

File System Performance

File System performance with respect to BREW can be measured by running BREWStone in the BREW PEK. For more information on performance measurement with BREWStone, see the *BREW Porting Evaluation Kit User Guide*.

BREW Services: Heap

This section describes the IHeap interface and functions for managing heap. It tells you how to use heap with BREW extensions and how to detect memory leaks. It touches on allocating memory in the system context and low memory notifications.

The Brew Heap is the heap memory region/RAM on a device for loading and running BREW applications. 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.

IHeap interface and functions

The function, OEM_GetinitHeapBytes() in OEMHeap.c, should be used by OEMs to configure the size and location of the BREW supervisor and user mode heaps.

BREW exposes the IHeap interface with APIs to be used for the dynamic allocation/de-allocation of memory and for obtaining information about heap usage. BREW also provides simple helper macros, for example MALLOC(), similar to the standard C library functions. For more information on the APIs and helper macros, see the *BREW API Reference Online Help*.

In addition to IHeap and the helper macros, OEM's can make use of OEM_Malloc(), OEM_Free () and so forth in static BREW applications/extensions. See xxx (Need reference).

BREW Heap Management

To prevent BREW applications from causing memory leaks, BREW keeps track of memory allocated by BREW applications. BREW maintains an information block called a heap node for each chunk of memory allocated on the BREW heap. In this heap node is a 32-bit ID, which is located 4 bytes ahead of the address returned from a memory allocation function call, for example, 4 bytes ahead of the return address/value of a successful call to MALLOC().

Dynamic BREW applications call MALLOC()/REALLOC() or IHEAP_Malloc()/ IHEAP_Realloc() to allocate memory. By default, BREW tags the allocated memory's heap node with the calling BREW application's module context 32-bit ID value.

NOTE: At BREW initialization, when existing BREW application's MIF files on a device are enumerated, BREW associates the applications with a particular module. A module can consist of one or more BREW applications. A BREW application runs in its own context known as the application context. Each application context corresponds to a module context. See Module B and associated Applets B1 and B2 in the diagram, How memory is marked on page 95. The 32-bit ID that BREW uses to track an applet's memory is actually a per-module ID. This means that 2 applets that come from the same module will have the same 32-bit ID.

Static applications, in addition to MALLOC()/ REALLOC() or IHEAP_Malloc()/ IHEAP_Realloc(), can also use OEM_Malloc()/OEM_Realloc() and sys_malloc()/sys_realloc(). For these calls, memory is always tagged under system context. Static applications can also choose to change the context to system context before calling MALLOC()/REALLOC() or IHEAP_Malloc()/ IHEAP_Realloc(). This would change the default behavior and cause the allocated memory to be tagged under system context.

When an application exists, BREW sifts through the heap looking for heap nodes marked with the application's 32-bit ID. On finding any such heap node, BREW calls OEMOS_BreakPoint() to report the memory leak and then frees the corresponding allocated memory automatically. BREW defers memory leak detection until the last application in a module exits. When this occurs on the BREW Simulator, an error is reported; on target, the leak is silently cleaned up.

NOTE: BREW will not call a destructor such as IInterfaceName_Release() or MyClass_Release() if the heap node is a memory allocation for a BREW Interface or an extension class respectively. An example would be if an application invoked ISHELL_CreateInstance() or MyClass_CreateInstance() but forgot to call IInterfaceName_Release () or MyClass_Release ().

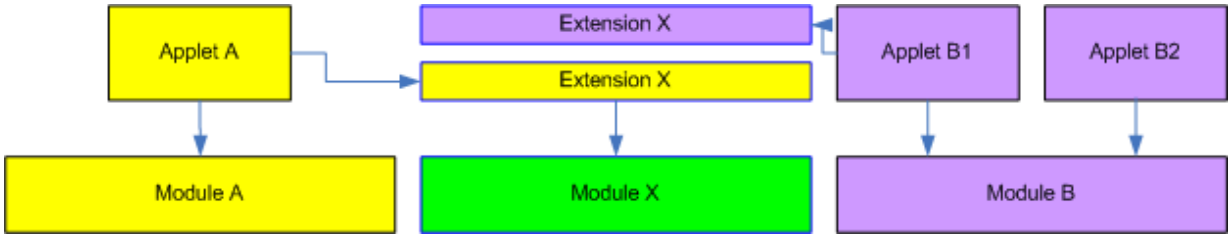
BREW Heap usage with BREW Extensions

When a BREW application creates an instance of an extension, it instantiates an IModule for that extension. See Module X in the diagram, How memory is marked on page 95. The memory allocated during instantiation of an extension's IModule is associated with the extension's module context. For example, this memory is tagged with the extension module context's 32-bit ID value.

After the initial instantiation of IModule, any other memory allocated for the instance of the extension is associated with the calling of the BREW application's module context. For example, this memory is tagged with the calling of the BREW application's 32-bit ID value. See Extension X in the diagram, How memory is marked on page 95

The following diagram illustrates how memory for the above mentioned configurations is marked. Each color represents a different 32 bit ID or a group of memory that will automatically be freed by BREW when the last user of the module disappears.

How memory is marked



Allocating memory in the system context

When a static extension's interface is meant to have a single instance and be shared among applications, or if it exposes an underlying single resource, for example, decoded information in a buffer, the class instance memory should be allocated in the system context. If the extension is allocated in a system context, some care needs to be taken to ensure a destructor is called to clean the resources. This can be done by linking multiple system objects to the application and creating the instance of the extension to cleanup it's references and/orto force a cleanup on AEE_Exit(). You can register for a system callback when AEE_Exit is completing to forcefully release the object and re-initialize any pointers or global object data

If an extension exposes a service, so that each calling application will require it's own data to be present, it is usually best for memory to be allocated in the context of the calling application’s module and be allocated once per the MyExtension_New() call. If possible, it is favorable in these situations to have the application allocate the memory and pass the buffer in to be filled. If a size determination is needed for the application, exposing an allocation measuring method is advisable. This can be done in a separate API or with certain overloaded parameters to the API that fill the memory. To guard against a resource leak, it is best to tie these instances to the executing application by using AEE_LinkSysObject() to ensure that the destructor is called when the module owning the applet is released.

Detecting Memory Leaks

BREW applications causing memory leaks can be detected when the module that owns the application is unloading.

Following is a case study that describes the debugging process. Consider the following :

```
App A owned by Module Z
App A has some MALLOC memory for a string that it forgets to FREE
App A has some MALLOC memory for general file binary data to be read
into that it forgets to FREE
App A has Interface N which is not LinkSysObject that it forgets to
release
App A has Interface O which is linked to this app with Link SysObject
that it forgets to release

At the time of App A closure and Module Z unload, set a breakpoint in
OEMOS_Breakpoint() function. See OEMOS.h for a list of AEEBRK_*** types
reported.
```

First

OEMOS_Breakpoint is called with AEEBRK_IFACELEAK for Interface O, and the AEEAppLeak structure contains further information about the leak.

Using this information, you can see who it belongs to by string.

```
AEE_GetMemGroupName(pal->dwMemGroup, szBuff, sizeof(szBuff));
```

This may return AppA or some numbered string for a downloaded applet such as 1052.

A member of AEEAppLeak provides a pointer to the buffer that leaked, so you can dump the buffer or inspect it to try and recognize it. These are usually convenient to recognize by looking at the first 4 bytes in a data.list as it is usually a Vtable for the interface that was failed to be freed. The next step would be to check that the SysObject's Callback is called, so that the destructor has a chance to clean up properly.

Second

OEMOS_Breakpoint is called with AEEBRK_MEMLEAK for the allocation of interface N, if this interface used MALLOC in module context and NOT in system context. It will not be similar to an interface O, since it looks just like an allocation not a SysObject. AEEAppLeak structure which contains further information about the leak.

Using this information, you can see who it belongs to by string.

```
AEE_GetMemGroupName(pal->dwMemGroup, szBuff, sizeof(szBuff));
```

This may return AppA or some numbered string for a downloaded applet such as 1052. An Extension interface does not own memory, but works off the calling application's memory. (That is unless AEE_EnterAppContext() was used before the allocations to modify the owner to system or otherwise).

A member of AEEAppLeak provides a pointer to the buffer that leaked, so that you can dump the buffer or inspect it. Buffers are usually easy to recognize by looking at the first 4 bytes in a data list since it is usually a Vtable for the interface that failed to be released.

Third

OEMOS_Breakpoint is called with AEEBRK_MEMLEAK for allocation of the string and then called again with the binary data. The AEEAppLeak structure contains further information about the leak.

Using this information, you can see who it belonged to by string.

```
AEE_GetMemGroupName(pal->dwMemGroup, szBuff, sizeof(szBuff));
```

This may return AppA or some numbered string for a donwloaded applet such as 1052.

A member of AEEAppLeak provides a pointer to the buffer that leaked, so that you can dump the buffer or inspect it. Buffers are convenient to recognize by looking at the data in a data.dump. For example, the string would probably be recognizable. The file binary data may be recognizable by someone who has familiarity with this applet and the binary look of the data, which the applet may load.

Low memory notifications

BREW applications can register for low memory notifications in BREW 3.1 and customize the behavior of freeing up memory. Low memory notifications are generated when a running BREW application's call to MALLOC fails and encounters insufficient memory.

Applications can call ISHELL_OnLowRam() or ISHELL_RegisterSystemCallback() with nSCBType as AEE_SCB_LOW_RAM, and register a callback function that will be called when the system reaches a low memory condition. This allows the application to do memory cleanup so as to free-up memory for the system.

Applications can also call ISHELL_OnLowRAMCritical() or ISHELL_RegisterSystemCallback() with nSCBType as AEE_SCB_LOW_RAM_CRITICAL to register for a callback to be invoked when the system reaches a low memory condition and sufficient RAM has not been gathered by AEE_SCB_LOW_RAM callback. This allows the application to do more memory clean up. The calling application must belong to the AEEGROUPID_LOW_RAM group to successfully register for the callback.

On devices with BREW applications registered to receive low ram notifications, BREW will first send out a low RAM notification to the applications who have registered for this condition. The expectation is that the registered application would then free up some resources to make more heap available. After this notification, BREW checks the available heap space again. If the required memory is still not available, then BREW will send out a critical low RAM notification to the applications who have registered for this condition. The expectation is that the registered application would then free up some resources/memory to make more heap available.

If no application has registered for the low RAM or critical low RAM notification, or even after sending the notifications, the amount of memory freed by the registered applications is insufficient, BREW will attempt to free up additional heap on its own. The way it does this is by unloading applications in the application stack.

BREW starts with the first suspended application that is below the current top visible application and works its way down the stack. It unloads the first application from memory (the application ends up getting an EVT_APP_STOP); however, the application history stack entry for this application is left untouched. This allows BREW to automatically restart the application when the current top visible application quits.

After unloading one application, BREW checks the available free memory again. If there is enough to satisfy the MALLOC request, BREW returns successfully from MALLOC. If there is still not enough memory, BREW unloads the next application in the stack, checks memory again and so on, until it gets enough memory to satisfy the MALLOC request. If after unloading all applications (except the top visible application), there is still not enough memory, BREW will return NULL for the call to MALLOC.

The stack for background applications is left untouched in this whole process. When the top visible application quits, BREW reloads the suspended application that it first unloaded and sends the application a resume event. The resumed application can retrieve any data that it saved in the application history stack at this time, since BREW preserved the AppHistory entry when the application was unloaded.

BREW Services: Pre-loading applications

This section explains the difference between pre-installed and static BREW applications and how to pre-install BREW applications. It mentions hybrid BREW applications and how to treat them.

Pre-loading BREW Applications

Pre-loaded BREW applications can be either dynamic or static. Pre-loaded dynamic BREW applications are referred to as pre-installed BREW applications.

Key differences between pre-installed and static BREW applications

The following table outlines some of the key differences between pre-installed BREW applications and static BREW applications.

Pre-installed (dynamic) BREW applications	Static BREW applications
Must be signed by QUALCOMM Internet Services (QIS) Product Support as a pre-installed application.	Do not need to be signed.
The request for pre-installed applications must originate from the carrier.	The carrier does not need to be involved in a static application pre-load.
Pre-installed applications can be removed using the AppManager's Application Management menu unless the application is protected.	Static applications cannot be removed.
Pre-installed applications can be disabled (auto-disabled in AppManager 2.X and through the Application Management menu in 1.X) unless the application is protected.	Static applications cannot be disabled.
Pre-installed applications can be upgraded through the AppManager's Application Management menu unless the application is protected. If the pre-installed application is protected, the pre-installed application can only be upgraded using MobileShop. A new version of the same application (same ClassID) can be downloaded over the air. The new application will then replace the protected pre-installed application.	Static applications can be upgraded using MobileShop. If a new version of the same application (same ClassID) is downloaded over the air, the dynamically downloaded application takes precedence over the statically downloaded version of the application.
Pre-installed applications can be recalled.	Static applications cannot be recalled.
Pre-installed applications reside in the File System on the device.	Static application resides in code space. Space occupied by static applications cannot be reclaimed.
Pre-installed applications' license information can be updated.	Static applications have no license information unless the application is upgraded over the air using MobileShop.

Initiating BREW application pre-load

Either the carrier or the OEM may be interested in pre-installing BREW applications.

Carrier-initiated pre-load

If a carrier is interested in pre-installing a BREW application, the carrier will approach the OEM and provide the OEM with or inform the OEM of the following:

- A signed pre-install BREW application.

- Whether the application is protected. Protected applications cannot be managed from the AppManager and cannot be deleted or disabled. They can, however, be upgraded and recalled.
- Whether the AppManager should be modified to prevent deletion and disabling of the pre-installed application.
- Whether the application should be launched from the AppManager or the native UI.

Carrier-initiated pre-loaded applications should be installed as pre-installed (dynamic) BREW applications.

OEM initiated pre-load

If the OEM wants to pre-load BREW applications onto a device, there are two possible options: pre-loading them as static applications and as dynamic applications.

Static BREW applications

Static BREW applications are built into the device software image and reside in code space.

An advantage of pre-loading static applications is that they do not need to be signed, so requests for pre-loaded static applications do not need to go through the carrier to get the application package.

Pre-installed (dynamic) BREW applications

Pre-installed BREW applications need to be signed (must have a BREW signature file) and packaged as pre-install applications by QIS Product Support before they can be loaded onto devices. QIS Product Support signs the pre-install application at the request of the carrier. OEMs cannot initiate requests for signed pre-install application packages from QIS. OEMs must channel these requests through the carrier instead. The following pre-install specifications are provided along with the pre-install package:

- The application is protected. Protected BREW applications have a MOD_PROTECTED flag set in the application MIF tail. The AppManager will not select protected applications for auto-disable and will not manage protected applications in its Application Management menu.
- The AppManager should be modified to prevent deletion and disabling of the pre-installed application.

- The pre-installed application should be launched from the AppManager or the native UI.

NOTE: A Registering Applications message may be displayed to the user on either phone power up or when the user first launches the pre-installed application. This message is displayed when an acknowledgement for the pre-installed application is sent to the Application Download Server (ADS).

Loading BREW pre-installed applications

Prerequisites

The following are prerequisites for loading BREW pre-installed applications:

- The BREW version of the pre-install application should match the BREW version on the device. The major and minor release numbers (BREW X.Y) of the two BREW versions should match.
- The BREW application should be signed as a BREW pre-install application, and the pre-install specifications should be provided.

BREW application loading process

Pre-loading of dynamic applications can be integrated into the build flashing process. OEMs should modify their product support tool (PST) to include the pre-loaded applications as part of the flashing process. The modification of the PST is specific to each OEM, so it is not described here.

R-UIM-based devices

License failure or application deletion of pre-loaded applications has been found on devices based on removable user identity modules (RUIM) in which BREW can run without an RUIM card. If there is an OEM-initiated pre-loaded application on a RUIM device, ensure that AEEGROUPID_ANY_SID is set as one of the dependencies in the MIF of the pre-installed application.

NOTE: On BREW 2.0 devices, the BREW 2.0 Porting Kit Patch 14 must be applied. For BREW 2.1 devices, BREW 2.1.2 or later is required.

Pre-install application as a native application

An OEM can choose to install the application as a native application so that it can be launched from the native menu instead of the Application Manager. The OEM should create an entry point (for example, an icon or menu item) in the device's native menu for this application and call ISHELL_StartApplet() with the application's ClassID to launch the application.

To prevent the AppManager from displaying the application, the AppManager code should be modified so that the application is:

- Not enumerated in the Application Launcher menu.
- Not enumerated in the Application Management menu.

On a BREW device, the application may get disabled. When an application is disabled, it needs to be restored (the application files need to be brought down from the ADS) before it can be launched. The following are ways in which a pre-installed application may get disabled:

- The application is signed as not protected when the user chooses to disable the application to claim space for downloading another application.
- The application can be upgraded over the air, and the download is canceled while the upgrade is in process.
- Versions of the AppManager prior to BREW 2.0 allow user-initiated disabling of applications.

NOTE: If the pre-installed application is selected to be launched from a native application when it is disabled, a dialog prompting the user to restore the application would need to be displayed.

Displaying the restore dialog from a native application

The following code can be used to determine whether the application is disabled or not before it can be started:

```
IShell    * pIShell = AEE_GetShell();
IDownload * pIDownload = NULL;
AppModInfo* pai = NULL;
DLITEMID    itemID;
if (pIShell) {
   ISHELL_CreateInstance(pIShell, AEECLSID_DOWNLOAD,
(void**)&pIDownload);
   itemID = ISHELL_GetClassItemID(pIShell, CLASS_ID_PREINSTALL);
```

```
   if (pIDownload) {
      pai = IDOWNLOAD_GetModInfo(pIDownload, itemID);
      IDOWNLOAD_Release(pIDownload);
      if (pai && pai->bRemoved) {
#error Application has been disabled. Display the restore dialog
prompting
#error the user if the application needs to be restored.
         return;
      }
   }
   // Launch the application
   ISHELL_StartApplet(pIShell, CLASS_ID_PREINSTALL);
}
```

If the disabled application needs to be restored, then the following code can be used to launch MobileShop to restore the application:

```
IShell    * pIShell = AEE_GetShell();
DLITEMID    itemID;
char        szBuffer[32];
if (pIShell) {
   itemID = ISHELL_GetClassItemID(pIShell, CLASS_ID_PREINSTALL);
   MEMSET(szBuffer, 0, sizeof(szBuffer));
   SNPRINTF(szBuffer, sizeof(szBuffer), "cmshop:Restore=%u", itemID);
   ISHELL_BrowseURL(pIShell, szBuffer);
}
```

Displaying restore dialog from AppManager

The pre-installed application to be launched from a native application can be launched from inside the BREW AppManager, letting the AppManager handle the disabled scenario.

```
The ClassID of the pre-installed application to be launched from the
AppManager is passed as an argument to AppManager as follows:
IShell    * pIShell = AEE_GetShell();
char        szBuffer[32];
if (pIShell) {
   MEMSET(szBuffer, 0, sizeof(szBuffer));
   // Additional arguments can be passed to AppManager and HandleEvent
   // routine in AppManager would need to be modified to parse
accordingly.
   SNPRINTF(szBuffer, sizeof(szBuffer), "LaunchID=%u",
CLASS_ID_PREINSTALL);
   ISHELL_StartAppletArgs(pIShell, AEECLSID_APPMANAGER, szBuffer);
}
```

The following changes are needed in AppManager to handle the launching of the application andto display the restore dialog if disabled.

1. Add a new state to AppMgr.h as follows:

```
#define ST_LAUNCHAPPVIEW       28
#define ST_MAX                 (ST_LAUNCHAPPVIEW + 1)
```

2. Add the new callback routine to the AppMgr structure used to schedule a close on itself.

```
  AEECallback       m_cbCloseSelf;
```

3. Add the prototype for the new routine in AppMgr.h as follows:

```
static boolean AppMgr_LaunchAppView(AppMgr * pme);
static void    AppMgr_CloseSelf(AppMgr *pme);
```

4. Add an entry for the new state added in *AppMgr_Init()* as follows:

```
SET_STATE(pme,ST_LAUNCHAPPVIEW,AppMgr_LaunchAppView,NO_ENTRY,FALSE)
;
```

5. Handle the arguments passed into the AppManager with the ClassID of the application to be launched. Modify the *AppMgr_HandleEvent()* as follows:

```
case EVT_APP_START:
{
   AEEAppStart *pArgs = (AEEAppStart *)dwParam;
   if (parg && parg->pszArgs && STRBEGINS("LaunchID=", parg-
  >pszArgs)) {
      AEECLSID    clsId;
      AppMgrItem *pApp;
      clsId = STRTOUL(parg->pszArgs + STRLEN("LaunchID="), NULL, 10);
      pApp = (AppMgrItem*)MALLOC(sizeof(AppMgrItem));
      if (pApp) {
         pApp->cls = clsId;
         pApp->dwItemID = ISHELL_GetClassItemID(pme->a.m_pIShell,
         clsId);
         pme->m_wState = ST_LAUNCHAPPVIEW;
         pme->m_pCurrentApp = pApp;
       // Handle Suspend / Resume of AppMgr so as to close it on resume
         pme->m_bResume = TRUE;
         // Do not free up and restore the cache on suspend / resume
         pme->m_bRetain = TRUE;
         // Free the Current App pointer on AppMgr exit
         pme->m_bFreeCurrentApp = TRUE;
         AppMgr_LaunchCurrentApplet(pme, FALSE);
      } else {
         return FALSE;
      }
    } else
      AppMgr_Start(pme);
  return TRUE;
}
```

6. Cancel the close self callback during the clean up in the AppMgr_Free() routine:

```
   CALLBACK_Cancel(&pme->m_cbCloseSelf);
```

7. Cancel the close self callback when the AppManager is getting suspended in the AppMgr_Suspend() routine:

```
static boolean AppMgr_Suspend(AppMgr* pme)
{
  // Don't handle if this is not the result of AppManager delegating
control
  // to hidden MShop applet
   if (!pme->m_bResume)
      return FALSE;
   CALLBACK_Cancel(&pme->m_cbCloseSelf);
```

8. Add the new routine AppMgr_LaunchAppView(), which would get called when AppManager is resumed and would close the AppManager.

```
static boolean AppMgr_LaunchAppView(AppMgr * pme)
{
   FreePtr((void**)&pme->m_pCurrentApp);
   pme->m_bResume = TRUE;
   CALLBACK_Init(&pMe->m_cbCloseSelf, AppMgr_CloseSelf, pMe);
   ISHELL_Resume(pMe->a.m_pIShell, &pMe->m_cbCloseSelf);
   return TRUE;
}
```

9. Add the new routine AppMgr_CloseSelf() that would close the AppManager.

```
static void AppMgr_CloseSelf(AppMgr *pme)
{
   ISHELL_CloseApplet(pme->a.m_pIShell, FALSE);
}
```

Pre-installed application deletion, disable, upgrade, and recall

The AppManager displays pre-installed applications in the Application Management menu (unless applications are protected) with an option to initiate removal. If desired, an OEM can customize the AppManager to prevent the Remove menu option from appearing in the Application Management menu. Following is an example:

```
// AppMgr_AppInfo() in AppMgr.c
if(pme->m_pCurrentApp->cls !=  CLASS_ID_PREINSTALL) {
```

```
    //CLASS_ID_PREINSTALL is defined to be the Class ID of the pre-
   installed app
    ai.wItemID     = IDC_REMOVE;
    ai.wText       = (pmi && (pmi->li.pt == PT_SUBSCRIPTION)) ?
IDS_CANCEL_SUBSCRIPTION : IDC_REMOVE;
    ai.wImage      = IDB_MGMT_REMOVE;
    IMENUCTL_AddItemEx(pme->m_pMenu, &ai);
}
```

AppManager version 2.X supports auto-disable. Protected applications will not be selected by BREW for auto-disable. Versions of the AppManager prior to BREW 2.0 allow the user-initiated disabling of applications. The code must be modified (similar to the above) to ensure that the disable option is not presented in the AppManager's Application Management menu.

Protected applications cannot be managed by the AppManager and cannot be deleted or disabled.

All pre-installed applications can be upgraded over the air and recalled regardless of whether they are designated as protected.

NOTE: If an application is pre-installed as a native application and the AppManager is modified to prevent enumeration in the Application Management menu, the application cannot be removed from the device.

Resolution verification

To verify if an application has been successfully pre-installed

1. Launch the AppManager and ensure that the application is enumerated and can be successfully launched, provided the application is intended to be launched through the AppManager and not the native UI.
2. If verifying on an RUIM-based device, use a valid RUIM card. Other than that, all the test procedures are the same as on a non-RUIM device.
3. If the application is protected, ensure that the application is not enumerated by AppManager's Application Management menu.
4. If the AppManager was modified to prevent deletion/disable of the pre-installed application, ensure that deletion and disable options are not available in the AppManager's Application Management menu.

5. If the application was pre-installed as a native application, ensure that the application is not enumerated by the AppManager, is not displayed in the AppManager's Application Management menu, and can be launched from the native UI.
6. If the pre-installed application is launched from the native UI, ensure that the restore dialog is displayed when the application gets disabled. Selecting the restore option from the dialog should result in MobileShop being launched to download and restore the application.

Static BREW applications

Pre-loaded static BREW applications do not need to be signed, so requests for pre-loaded static applications do not need to go through the carrier. Static BREW applications are built into the device software image and reside in code space.

NOTE: For BREW versions 3.1.0 and 3.1.1, the dynamic version of an application downloaded over the air will override the static version of the same application (the same ClassID) only if the dynamic version of the application has the same MIF filename as the static version of the application. Any application downloaded over the air has a numbered MIF name, for example, 400.mif. As a consequence, if an OEM plans to do an over-the-air upgrade of an application that is being statically loaded on the phone, it is strongly recommended that this application be signed by QUALCOMM as a pre-installed application. This signed pre-installed application can then be statically loaded on the phone by the OEM. Starting with BREW version 3.1.2, this issue is fixed so that a dynamic version of an application will always override the static version of the same application (the same ClassID), regardless of whether or not their MIFs' names are identical.

Loading a BREW application as a static application

Prerequisites

Observe the following prerequisites:

- The BREW version of the application should match the BREW version on the device. The major and minor release numbers (BREW X.Y) of the two BREW versions should match.
- OEMs must obtain the source code for the static application from the application developer.

- The ClassID for the static BREW application should be generated using the ClassID Generator tool available on theBREW OEM Extranet.

BREW application loading process

The procedure for pre-loading BREW applications as static applications is detailed in the advanced section Static Modules on page 183. BREW 3.1 and above require static applications to have MIFs. The MIFs may be either constant (linked into the core software image) or dynamic (in EFS) MIF files.

Constant MIFs are upgradable (fixed = FALSE) by default. This setting allows them to be upgraded over the air and should not be modified.

Pre-install application as a native application

An OEM can choose to install the application as a native application so that it can be launched from the native menu instead of the AppManager. The OEM should create an icon in its native menu for this application and call ISHELL_StartApplet() with the application's ClassID to launch the application.

To prevent the AppManager from displaying the application, the AppManager code should be modified so that the application is not enumerated in the Application Launcher menu

Static BREW application deletion, disable, upgrade, and recall

Static BREW applications cannot be deleted, disabled, or recalled. Static BREW applications are not enumerated in the Application Management menu of the AppManager, because they do not have a valid download item ID. Consequently, users cannot check for upgrade versions of the applications from the AppManager.

Static applications can be upgraded using MobileShop. If a new version of the same application (same ClassID) is downloaded over the air, the dynamically downloaded application takes precedence over the statically downloaded version of the application. The file system space occupied by the statically loaded application cannot be reclaimed.

NOTE: Static applications cannot be removed from the device. The file system space occupied by the statically loaded application cannot be reclaimed.

Resolution verification

To verify if the static application has been properly pre-loaded

1. Launch the AppManager and ensure that the application is enumerated and can be successfully launched, provided that the application is intended to be launched through the AppManager and not the native UI.
2. If the application was pre-loaded as a native application, ensure that the application is not enumerated by the AppManager's Application Launcher menu.
3. Ensure that the application does not appear in the AppManager's Application Management menu.

Hybrid BREW applications

A hybrid BREW application contains both dynamic modules and static ones. The two different types of modules should be handled separately. For the dynamic modules of the application, OEMs should treat them as dynamic pre-loaded applications and use the instructions specified in BREW pre-installed (dynamic) applications. For the static modules of the application, OEMs should treat them as static applications and refer to the procedures in Static BREW applications on page 108.

BREW Services: 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 $BREW command with a tag of AT, and the resulting response packet, including the tag, is ATOK. As a result, when AT$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. 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. 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). 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. 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. 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. 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: • ESCHEMENOTSUPPORTED (a BREW error code) • 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:Kbytes

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.

AMSS changes required to enable BREW SIO

QUALCOMM's AMSS 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 AMSS changes based upon 6050 chipset AMSS 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 AMSS 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 AMSS 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:

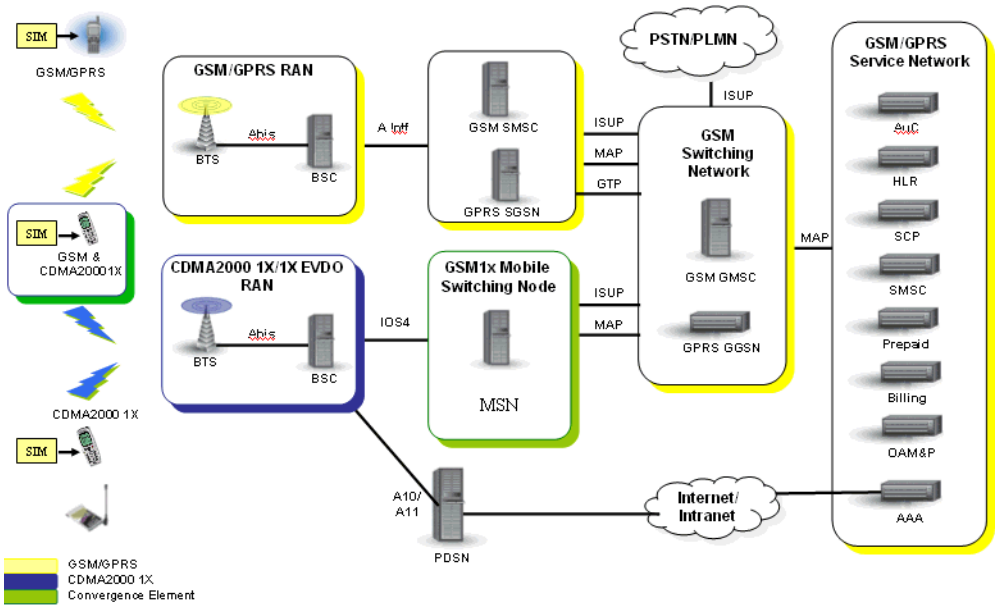
```
rdm_init_config_table();
```

BREW Services: 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 AMSS features must be present:

- Sockets
- File system
- TAPI
- Operating system

Prerequisites

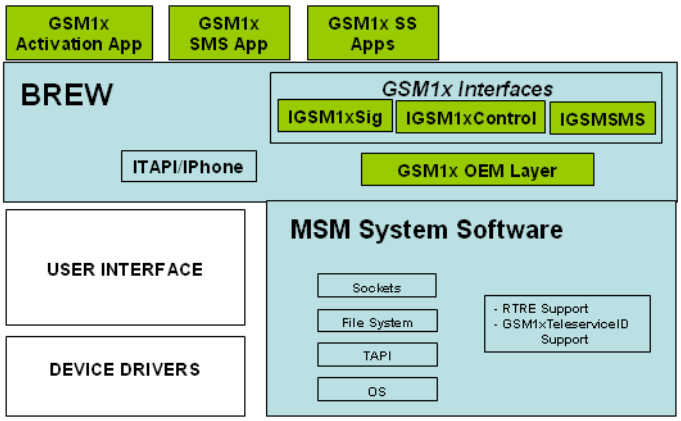
GSM1x developers need to familiarize themselves with the following.

- 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.

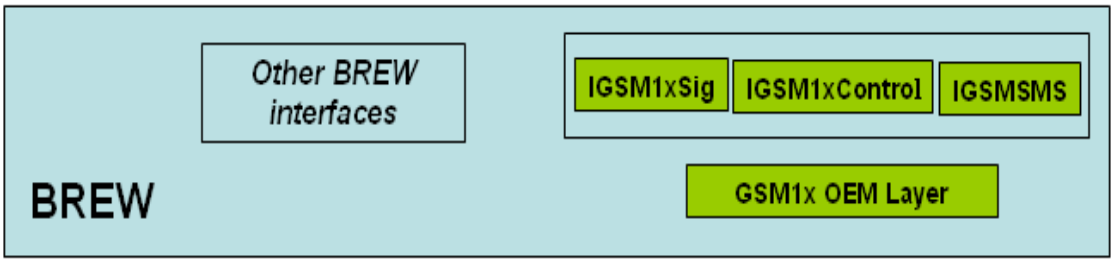
Understanding GSM1x BREW interfaces

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.

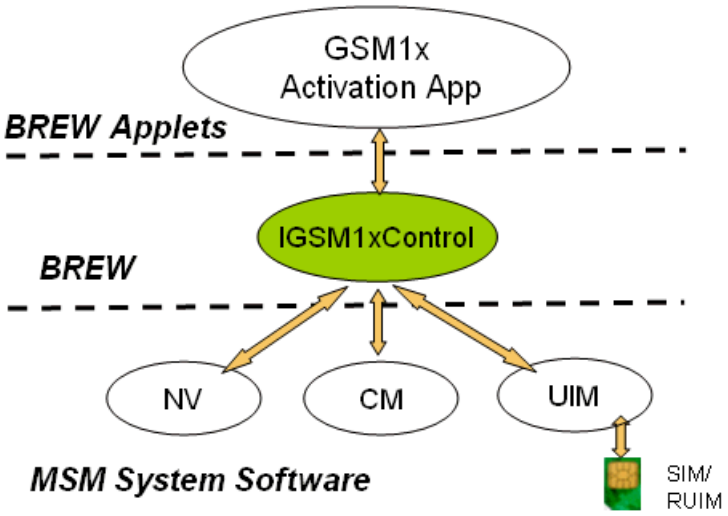
GSM1x BREW interfaces



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.

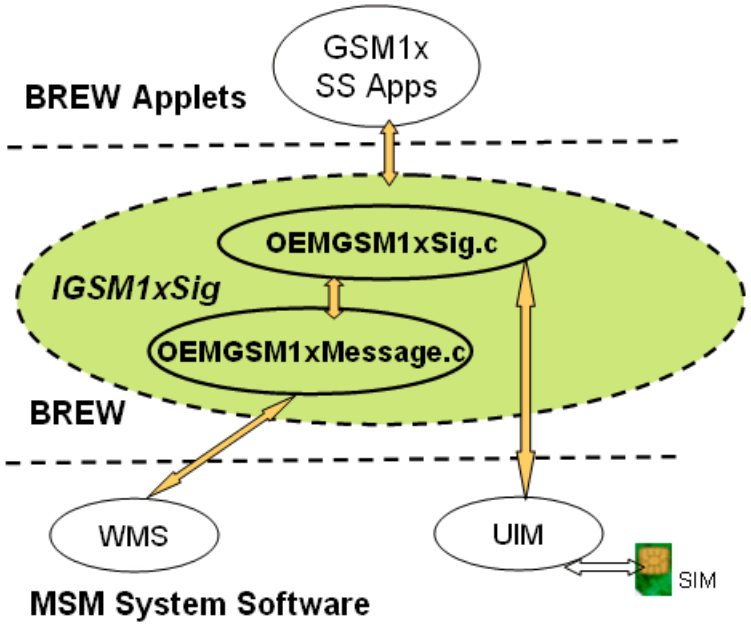
The following table shows source files and descriptions.

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 AMSS 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 AMSS 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.

Integrate BREW Tools Interface Layer (BTIL)

The BREW® Tools Interface Layer (BTIL) is a complete protocol suite for communicating with and remotely controlling a BREW-enabled device from a Windows-based PC over a serial port or other medium. It was designed to replace OEM-specific diagnostic DLLs and, in doing so, add additional features and flexibility for current and future versions of the BREW Tools Suite, Porting Evaluation Kit (PEK) Studio, and other BREW-related Windows software.

Features of the BTIL protocol

The following features are realized by using the BTIL protocol over the legacy method:

- Independence from the OEM-specific diagnostic protocols. BTIL is written using the BREW API, making heavy use of the BREW SIO (IPort) features to communicate. As such, it makes no requirements or assumptions that the device has a diagnostic protocol of its own to operate. Moreover, as the packet format is specific to BTIL, OEM-specific diagnostic protocols need not be exposed to the user. Because everything is done in BREW, OEM configuration is minimal.
- Enhanced device security. Currently, many device manufacturers cannot secure their vital diagnostic services, and this allows members of the general public access to areas of functionality that are not meant for them. Current attempts at protecting this often end up being an inconvenience for registered BREW developers. BTIL ships in a “locked” state in which it will not perform any operations until the BREW digital test signature for that phone has been uploaded to the device and verified by the device software. In this way, unauthorized users are kept out, while developers can use the tools with no real inconvenience.
- Easily upgraded. Because BTIL is implemented as a BREW applet, it can be easily upgraded as necessary by the user without interaction from the OEM. Tools provided with developers’ BTIL packages allow them to do this with a couple of clicks.

- Portability to new media types. BTIL on a device supports the same serial and USB connection methods that the existing technology does. BTIL was built with the Internet in mind and as such the underlying protocols make no assumptions about data integrity, packet ordering, and retransmission. Future versions of BTIL will support additional transport mechanisms, including UDP OTA, the Mobile Display Digital Interface (MDDI), Bluetooth, and Infrared Data Association (IrDA) communications.
- Reliable bi-directional communications. One of the serious limitations of the existing OEMLayer.dll is that BREW applets on the device cannot directly communicate back with the software on the Windows PC. With BTIL, both communications inbound to and outbound from the device are fully supported. In addition, extensions can make use of reliability layers to automatically perform any sequencing or retransmission necessary as required for that extension's data.
- User extensions: BTIL ships with three built-in extensions that contain all of the functionality in the legacy OEMLayer.dll files. If a user or OEM has a need for additional functionality beyond those provided in the built-in extensions, they are encouraged to write an extension of their own, using the same connection management and reliability mechanisms provided to the built-in extensions.

BTIL pre-installation rationale

Because BTIL is intended to be the primary mechanism by which developers load files onto a device in the future, it is important that BTIL be present on the device when it ships from the manufacturer.

A significant effort is being made to reduce the time it takes BREW developers to get up and running on devices. By pre-installing BTIL, OEMs remove any dependency on device-specific PSTs, dongles, or secret menus. This allows a developer to use a device purchased from a retailer after only obtaining a digital test signature from the QUALCOMM site.

BTIL resource usage

BTIL is designed to be relatively compact and to use resources only as required by the system. BTIL utilizes the AT$BREW command and the BREW Command Processor to instantiate the BTIL applets only when they are being connected to, and as such, no memory is used by BTIL except while PC tools are being used. The BTIL applets at the time of this writing require approximately 70 kilobytes of persistent storage space, which typically becomes a part of the device's image via the persistent file system.

Adding BTIL to Your Device

One of the primary goals of BTIL development was to make adoption of BTIL by the OEMs a straight forward process. Follow these procedures to add BTIL to your device.

1. Download the latest BTIL Development Kit. Unless directed not to do so by the instructions on the web site, download the latest version of the BTIL Development Kit. The BTIL client application is supported in BREW 3.1.4 or later releases of the BREW API; however, it will run on most 3.1.x BREW devices.

   NOTE: This development kit is the same one that is provided to BREW developers. There are no files beyond the ones provided in the kit and those provided in the BREW Porting Kit that you will need to pre-install BTIL on your device.

2. Ensure that FEATURE_BTIL is enabled. Check your OEMFeatures.h file and make sure that FEATURE_BTIL is turned on. This will cause the device to instantiate internal BREW libraries needed by the dynamic portion of the BTIL software.

3. Once feature FEATURE_BTIL is turned on, you may need to add the library aeebtil.lib to your link stage in your compilation process. aeebtil.lib is shipped with the BREW Porting Kit.

4. Add the BTIL applets to the persistent file system. To build the dynamic module statically, make these as persistent files using the BIN2SRC tool. For more information on the BIN2SRC tool, see the *BREW PK Utilities Guide*.

5. Following are the steps needed to make BTIL work on the Persistent File System. They should be integrated into the OEM's make/build system.

   a. Locate the Client BTIL modules. By default, this is in C:\Program Files\Common Files\Qualcomm\BTIL Development Kit\Client\device. This directory contains the BTIL dynamically loaded module (MOD) files and Module Information Files (MIFs) compiled for various types of compilers. Choose the directory corresponding to the compiler, processor, and build settings for your build, and open a command prompt window within this directory to execute the commands in the following step.

   b. Create a persistent file entry for the MIFs and MODs by executing the BIN2SRC tool, found in the utils directory of the BREW Porting Kit, on the MIFs and MODs to produce a source version of the MIF/MOD binary. From a command prompt, execute the following commands:

```
bin2src –sbtilmain\btilmain.mif -dfs:/mif/btilmain.mif
bin2src –sbtilsec\btilsec.mif -dfs:/mif/btilsec.mif
```

```
bin2src –sbtilmain\btilmain.mod -dfs:/mod/btilmain/btilmain.mod
bin2src –sbtilsec\btilsec.mod -dfs:/mod/btilsec/btilsec.mod
```

This creates files called btilmainmif.c, btilsecmif.c, btilmainmod.c, and btilsecmod.c. Integrate these files in your build.

c. Add the global variable declaration of MIF and MOD to OEMConstFiles.c as follows:

```
extern AEEConstFile gBTILMAIN_MIF;
extern AEEConstFile gBTILSEC_MIF;
extern AEEConstFile gBTILMAIN_MOD;
extern AEEConstFile gBTILSEC_MOD;
```

d. Add the address of gBTILMAIN_MIF, gBTILSEC_MIF, gBTILMAIN_MOD and gBTILSEC_MOD to the gpOEMConstFiles table in OEMConstFiles.c as follows:

```
static const OEMFSPersistentFile * gpOEMCosntFiles [] = { …
&gBTILMAIN_MIF, &gBTILSEC_MIF, &gBTILMAIN_MOD, &gBTILSEC_MOD, …
NULL};
```

Testing BTIL Integration

Follow these procedures to test BTIL integration:

1. Download the latest BREW Tools Suite. The BREW Tools Suite is available on the BREW Developer Extranet.
2. Install a test signature for the device. Test signatures can be obtained from the Test Signature Generator page on the BREW Developer Extranet. You will need to provide an ESN, an International Mobile Equipment Identifier (IMEI), or Mobile Equipment Identifier (MEID) for the device being connected to in order to obtain this signature. BTIL utilizes the same signatures used to sign applications being debugged on the device.
3. Once you have a test signature for the device, open BTIL's signature directory. This can quickly be done by accessing the BTIL folder under Start > Programs and by selecting the "Test Signature Repository" item. Copy the signature into this directory and rename the signature to its ESN or IMEI plus the .sig file extension. For example, for a device with ESN 0x1234ABCD, the signature file should be named 1234ABCD.sig.

4. Ensure that the data service is running on the port in use. BTIL is started by the host PC using the AT$BREW command. This requires that the port being used is configured to have the data (modem) service running on it.

5. Start the BREW Logger and connect to the device. From the Windows Start Menu, select Programs > BREW Tools Suite > BREW Logger. Connect the BREW Logger to the device by selecting the appropriate port with the BTILOEM tag beside it and by pressing the OK button. Once connected, select Configure > Start Logging. A new window should appear displaying log items. You can press keys on the device's keypad to generate log events.

6. If the keypress messages show up on the display, you have successfully integrated BTIL onto your device.

7. Run the PEK PCIT and SAT tests. For more thorough testing of your BTIL integration, run the PC Interface Test (PCIT) and Serial Access Test (SAT) portions of the Porting Evaluation Kit using BTILOEM as the interface DLL for the connection. BTIL compatibility was first introduced in PEK 3.1.4.

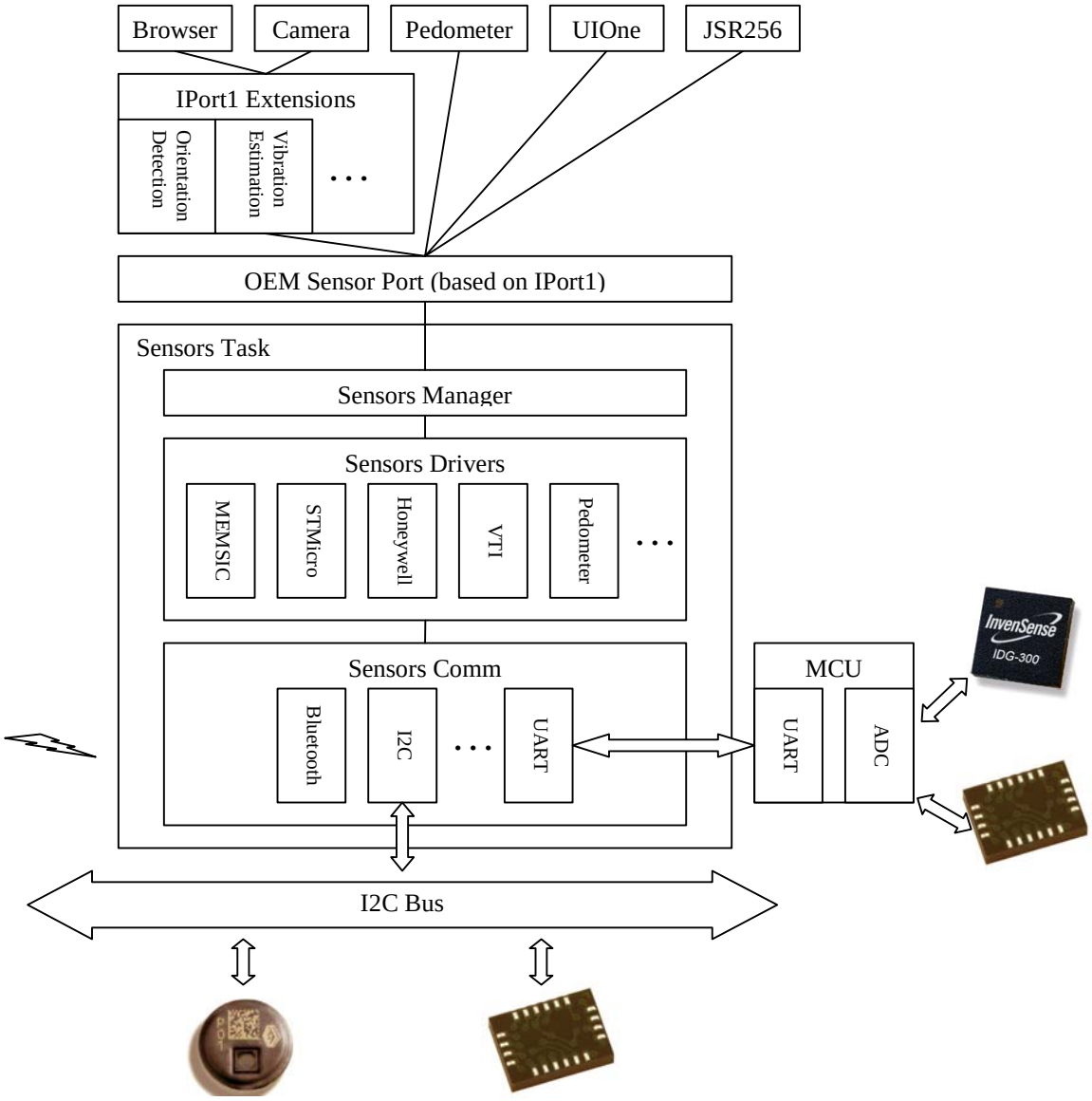
BREW Services: OEM Sensors

Overview

Brew exposes the IPort1 API, which provides applications with a generic interface for a bi-directional data stream that provides Read/Readable, Write/Writeable, Control interfaces. Since IPort1 is a generic interface, it must be created using a Class ID specific to its usage. OEM Sensors implements a specific class, AEECLSID_Port1Sensor, for Sensors.

Architecture

The Sensors system architecture is shown in this diagram.



Enabling the feature on the device

An OEM will need these files to enable Sensors in their environment:

- AEEPort1Sensor.bid, available as part of the Brew SP01 release
- OEMModTableExt_Sensors.h
- OEMSensorPort.c

- ext_sensors.min

Turning on Sensors

These are the list set of FEATURE defines needed to turn on Sensors:

- FEATURE_SENSORS. All the Sensors' code is compiled under this flag, so this feature must be enabled in the build.
- OEMModTableExt_Sensors.h. The entry point function is defined in OEMModTableExt_Sensors.h, so the OEMModTable must include this file and define the SENSORS_STATIC_CLASS_LIST_ENTRY. This code example shows how it is done in AMSS.

```
#ifdef FEATURE_SENSORS
#include "OEMModTableExt_Sensors.h"
#endif

And

#ifdef FEATURE_SENSORS
SENSORS_STATIC_CLASS_LIST_ENTRY
#endif
```

Dependencies

As shown in the architecture diagram, the sensors stack has the sensors manager and device drivers that the OEM functions rely on.

Extensibility

If OEM's wish to use the Sensor types exposed by the protocol, OEMs can use the class AEECLSID_Port1Sensor and the IPort1 interface functions to communicate with the sensors.

If OEMs wish to develop their own sensor types, they must

1. Contact QCT so that the new sensor type can be added. QCT will update the protocol document, add support in the sensors manager for the new data type, and write device drivers for the new data type.

2. If OEMs wish to attempt this process themselves, they need to add a new Class ID, write the device driver for the new data type, and contact QCT to update the protocol document and make it public so that support can be added in the sensors manager.

BREW Services: vCard/vCal support

•vCard is based on RFC 2426 and supports parsing vCard v2.1 and v3.0 data streams and generating vCard v3.0 data stream. vCal is based on RFC 2445 and supports vCal 1.0 and 2.0. The BREW OEM Porting Kit contains these pre-built persistent files (see Creating persistent files on page 86) and the BREW SDK contains the interface definitions:

- AEEIvParm.h
- AEEIvProperty.h
- AEEIvObject.h
- AEEIvCard.h
- AEEIvCalObject.h
- AEEIvObjectParser.h
- AEEIvCalStore.h
- AEEvAlarm.bid
- AEEvCalendar.bid
- AEEvCalStore.bid
- AEEvCard.bid
- AEEvDaylight.bid
- AEEvEvent.bid
- AEEvFreeBusy.bid
- AEEvJournal.bid
- AEEvObjectParser.bid
- AEEvParm.bid
- AEEvProperty.bid
- AEEvStandard.bid

- AEEvTimeZone.bid
- AEEvToDo.bid

Depending on whether you port vCardvCal or AEEIvCalStore, you will link these libraries with the device build:

- aeepdiparsers.lib
- aeevcalstore.lib

The PDIParsers module contains the stand-alone vCard/vCal parsing engine.

To port vCard/vCal

1. Link aeepdiparsers.lib withthe device build .

AEEIvCalStore

AEEIvCalStore allows BREW applications to add (both 1.0 and 2.0), update, get (2.0), and delete calendaring components (vEvent, vToDo, vJournal, vTimezone.). It contains a stand-alone implementation of the IvCalStore interface. The calendar data is stored in fs:/shared/Calendar.db. Implementation of IvCalStore requires ISQL, and porting is required for full integration. There are two porting options: Using BREW IvCalStore for all calendaring, and re-implementing IvCalStore.

To port AEEIvCalStore

1. Link aeevcalstore.lib with device build.
2. If applicable, re-work the native Calendar Application to use IvCalStore API for storing calendar entries.

To port AEEIvCalStorRe-Implement IvCalStore

1. Re-implement IvCalStore to store calendar entries to native storage.

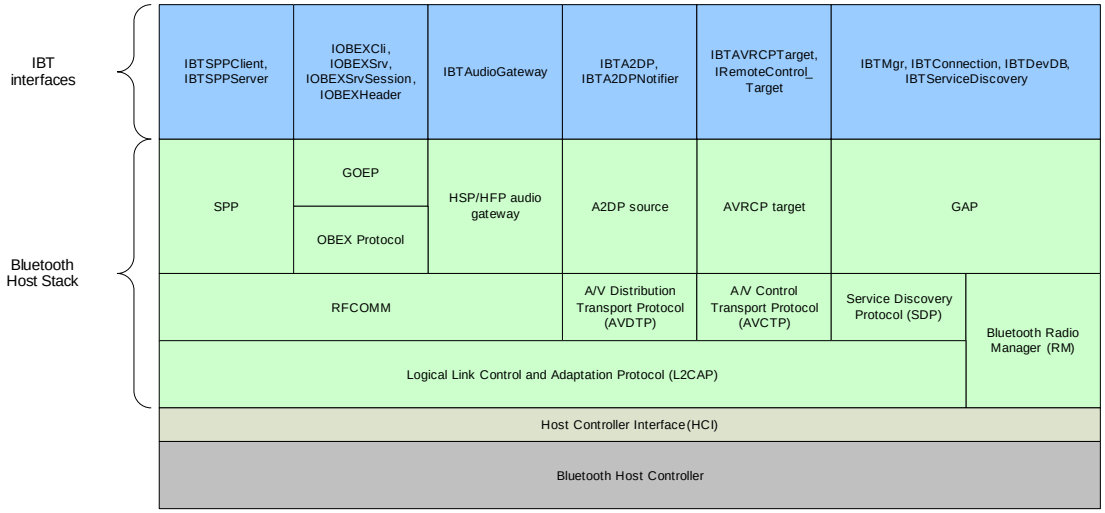
Overview

This section describes the IBT interfaces which support these Bluetooth profiles: Generic Access Profile (GAP), Serial Port Profile (SPP), Headset Profile (HSP), Handsfree Profile (HFP), Advanced Audio Distribution Profile (A2DP), Audio Video Remote Control Profile (AVRCP), and Generic Object Exchange Profile (GOEP). These profiles are supported by these interfaces:

- GAP: IBTMgr, IBTConnection, IBTDevDB, IBTServiceDiscovery
- SPP: IBTSPPClient, IBTSPPServer
- HSP and HFP: IBTAudioGateway
- A2DP: IBTA2DP, IBTA2DPNotifier
- AVRCP: IBTAVRCPTarget, IRemoteControl_Target
- GOEP: IOBEXCli, IOBEXSrv, IOBEXSrvSession, IOBEXHeader

Architecture

This diagram shows IBT interfaces as well as the underlying profiles and protocols.



Enabling the feature on the device

These files from the SDK are required:

- Header files
  - AEEBT.h
  - AEEBTCommon.h
  - AEEBTDef.h
  - AEEBTDevDB.h
  - AEEBTDevDBGetObject.h
  - AEEBTSDDef.h
  - AEEBTSPP.h
  - AEEBTSPPClient.h
  - AEEBTSPPClientGetObject.h
  - AEEBTSPPServer.h
  - AEEBTSPPServerGetObject.h
  - AEEIBTA2DP.h
  - AEEIBTA2DPNotifier.h
  - AEEIBTAudioGateway.h

  - AEEIBTAVRCPTarget.h
  - AEEIBTConnection.h
  - AEEIBTMgr.h
  - AEEIBTServiceDiscovery.h
  - AEEIOBEXCli.h
  - AEEIOBEXHeader.h
  - AEEIOBEXSrv.h
  - AEEIOBEXSrvSession.h
  - AEEIRemoteControl_Target.h
  - AEEOBEXComDefs.h
- BID files
  - AEEBT.bid
  - AEEBTA2DP.bid
  - AEEBTA2DPNotifier.bid
  - AEEBTAVRCPTarget.bid
  - AEEBTAudioGateway.bid
  - AEEBTConnection.bid
  - AEEBTMgr.bid
  - AEEBTServiceDiscovery.bid
  - AEEOBEXCli.bid
  - AEEOBEXSrv.bid
  - AEERemoteControl_AVRCP_1_0_Target.bid

The following files in the AMSS release are required:

- Header files
  - AEEBTUtils.h
  - AEEOBEXDefs_priv.h

  - OEMBT.h
  - OEMBTA2DP.h
  - OEMBTAG.h
  - OEMBTAVRCPTarget.h
  - OEMBTDevDB.h
  - OEMBTMgr.h
  - OEMBTSD.h
  - OEMBTSPP.h
  - OEMBTUtils.h
  - OEMBT_priv.h
  - OEMOBEXDefs.h
  - OEMOBEXDefs_priv.h
- Source files
  - AEEBT.c
  - AEEBTA2DP.c
  - AEEBTAG.c
  - AEEBTAVRCP.c
  - AEEBTConnection.c
  - AEEBTDevDB.c
  - AEEBTMgr.c
  - AEEBTSD.c
  - AEEBTSPPClient.c
  - AEEBTSPPServer.c
  - AEEBTUtils.c
  - AEEOBEXCli.c
  - AEEOBEXCommon.c
  - AEEOBEXHeader.c

  - AEEOBEXSrv.c
  - AEERemoteControl_Target.c
  - OEMBT.c
  - OEMBTA2DP.c
  - OEMBTAG.c
  - OEMBTAVRCPTarget.c
  - OEMBTDevDB.c
  - OEMBTMgr.c
  - OEMBTSD.c
  - OEMBTSPP.c
  - OEMBTUtils.c
  - OEMOBEXCli.c
  - OEMOBEXCommon.c
  - OEMOBEXHeader.c
  - OEMOBEXSrv.c

These AMSS features need to be defined on the device build to enable the interfaces:

- FEATURE_IBT, FEATURE_IBT_DYNAMIC
  - These two features enable the IBT interfaces.
- FEATURE_BT_SPP
  - This feature enables SPP in the Bluetooth host stack.
- FEATURE_BT_AG , FEATURE_BT_HFP_1_5 , FEATURE_AVS_BT_SCO_REWORK
  - These features are required for enabling HSP and HFP in the Bluetooth host stack.
- FEATURE_BT_EXTPF_GOEP
  - This feature enables GOEP support in the Bluetooth host stack.
- FEATURE_BT_EXTPF_AV

  - This feature enables A2DP and AVRCP in the Bluetooth host stack.
- FEATURE_BT_EXTPF
  - This is the global feature for profile support in the Bluetooth host stack.

Dependencies

The IBT interfaces have dependencies on these interfaces and URIs:

- Interfaces
  - IAddrBook
  - ICallHistory
  - ICallMgr
  - ICall
  - IModel
  - IQI
  - IShell
  - ISignal
  - ISignalBus
  - ISignalCBFactory
  - ISignalCtl
  - ITelephone
- URIs
  - “tel:Call?Number=<number>”
  - "tel:AnswerCall"
  - “soundman:select?dev=<device>”
  - “soundman:deselect?dev=<device>”

Overview

IBitmapFX is the interface, which is used to perform the image processing of bitmaps. It exposes the image processing layer functions to a BREW application and provides a BREW application interface to image processing functionalities such as posterize, solarize, and filter.

Enabling the feature on the device

These files from the SDK are required:

- Header files
  - AEEBitmapFX.h
  - AEEIBitmap.h
  - AEEIQI.h
- Bid File
  - AEEBitmapFX.bid

These files in the AMSS release are required:

- Header files
  - IYCbCr.h
  - ipl.h
  - YCbCr_priv.h
  - ipl_attic.h
  - ipl_compose.h
  - ipl_convert.h
  - ipl_downSize.h

  - ipl_efx.h
  - ipl_efx_images.h
  - ipl_helper.h
  - ipl_qvp.h
  - ipl_rotAddCrop.h
  - ipl_types.h
  - ipl_upSize.h
  - ipl_util.h
  - ipl_xform.h
- Source files
  - YCbCr.c
  - OEMIBitmapFX.c
  - ipl_attic.c
  - ipl_compose.c
  - ipl_convert.c
  - ipl_downSize.c
  - ipl_efx.c
  - ipl_helper.c
  - ipl_hjr.c
  - ipl_rotAddCrop.c
  - ipl_upSize.c
  - ipl_util.c
  - ipl_xform.c

Define FEATURE_IIPL to turn on this feature.

Dependencies

The IBitmapFX interfaces have dependencies on these interfaces:

- Interfaces
  - IQI
  - IShell
  - IModule
  - IBitmap

BREW Services: RMC insertion and removal notifications

Overview

This feature provides RMC insertion and removal notifications to the interested BREW applications (applications registered to receive the RMC insertion and removal notifications via the ISHELL_RegisterNotify() API) and involves an OEM layer implementation for the IDeviceNotifier API.

OEM layer changes are made in OEMDeviceNotifier.c and are based upon an implementation of the HotPlug Manager feature in EFS. The HotPlug Manager feature in EFS provides the notification API, so that interested tasks can be signaled (via callbacks) when there is a media event of interest like removeable media card (RMC) insertion or removal. The OEMDeviceNotifier.c file implements the functionality necessary to interface with the EFS Hot Plug Manager feature to receive the RMC insertion and notification events from EFS. These notification events are then sent to the application interested in receving these notifications.

Pre-requisites

BREW 3.1.5SP01 or later has this functionality implemented in the Porting Kit layer's OEMDeviceNotifier.c file.

EFS VU MSMSHARED_EFS.01.02.48 or higher implements the HotPlug Manager feature needed by this feature.

If an OEM uses a file system other than EFS, an equivalent of HotPlug Manager functionality of EFS must be present to enable this feature in BREW by modifying OEMDeviceNotifier.c to interface with the alternate file system chosen.

Implementation Details

A new nmask is added for MMC insertion and removal notification in AEEDeviceNotifier.h:

```
NMASK_DEVICENOTIFIER_MMC
#define NMASK_DEVICENOTIFIER_MMC          0x00000080
```

The MMC removal and insertion events are defined by following macros in AEEDeviceNotifier.h:

```
#define AEE_INSERT_MMC 0
#define AEE_REMOVE_MMC 1
```

See AEEDeviceNotifier.h for a detailed description of the nmask and insertion and removal macros.

When a removable media device is inserted or removed, the OEM layer notifies BREW of these events using the following macro (defined in OEMDeviceNotifier.h):

```
#define AEE_SEND_MMC_NOTIFY(p) {\
(void) AEE_Notify(AEECLSID_DEVICENOTIFIER, NMASK_DEVICENOTIFIER_MMC, p);}
```

See the OEMDeviceNotifier.h file for a detailed description of the macro and it's fields.

This feature is active in the OEM layer if FEATURE_EFS_HOTPLUG is defined in the build. This feature flag indicates that EFS provides the HotPlug Manager functionality needed for RMC insertion and removal notifications to work.

OEMDeviceNotifier.c details

The reference implementation only caters to the BREW supported MMC, specified via the AEEFS_CARD0_DIR (fs:/card0/) path. The default device mapping of this path is /mmc1.

AEEDeviceNotifier_New() calls CDeviceNotifier_HpmInit(). The CDeviceNotifier_HpmInit() function registers a callback handler with EFS for RMC insertion and removal notification using the EFS provided hotplug_notify_register() API. The callback function is registered in CDeviceNotifier_HpmNotify().

CDeviceNotifier_HpmNotify() is the callback function invoked from EFS when RMC insertion or removal is detected. This notification function checks that the event received is for the MMC card supported in the implementation, and then it schedules a BREW system callback via AEE_SYS_RESUME().

BREW system callback is processed by the following function, which calls AEE_SEND_MMC_NOTIFY to dispatch the notification event to interested applications.

```
static void CDeviceNotifier_HpmSysNotify(void* pv)

{
   CDeviceNotifier* pMe = (CDeviceNotifier*) pv;
   AEEDeviceNotify devNtfy;

   FARF(DEVNTFY,("CDeviceNotifier_HpmSysNotify"));

   if (pMe->m_lastEvent == HOTPLUG_NOTIFY_EVENT_CARD_INSERTED) {
      FARF(DEVNTFY,("AEE_INSERT_MMC"));
      devNtfy.wParam = AEE_INSERT_MMC;
   }
   else {  // this is fine. no other event can come here.
      FARF(DEVNTFY,("AEE_REMOVE_MMC"));
      devNtfy.wParam = AEE_REMOVE_MMC;
   }
   devNtfy.dwParam  = pMe->m_mmcId;

   AEE_SEND_MMC_NOTIFY(&devNtfy);
}
```

BREW UI: Display support

This section contains core display information and discusses the implementations of bitmaps, display, and TAPI.

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.

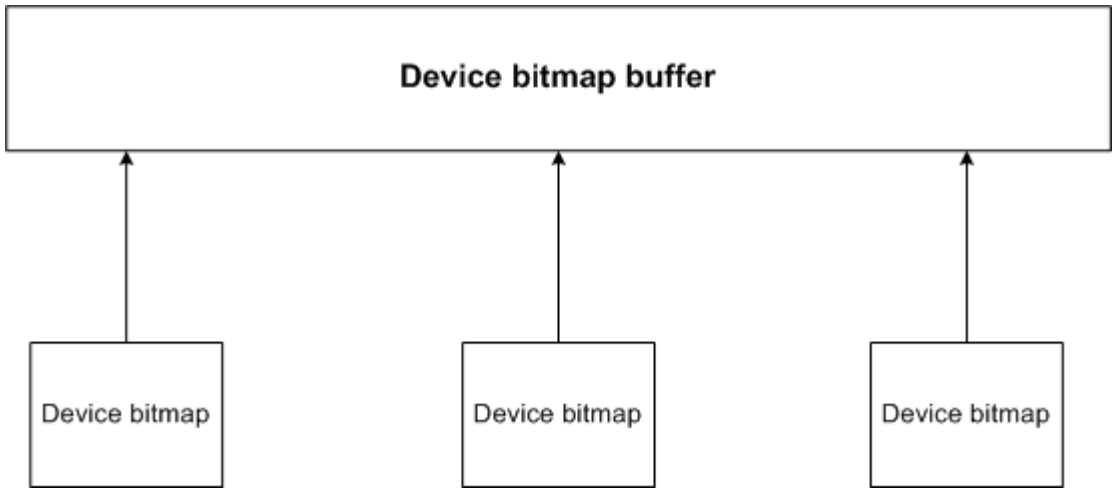
BREW accesses displays through the following three bitmap interfaces:

- 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_DEVBITMAP*n* where *n* is 1, 2, 3, or 4.

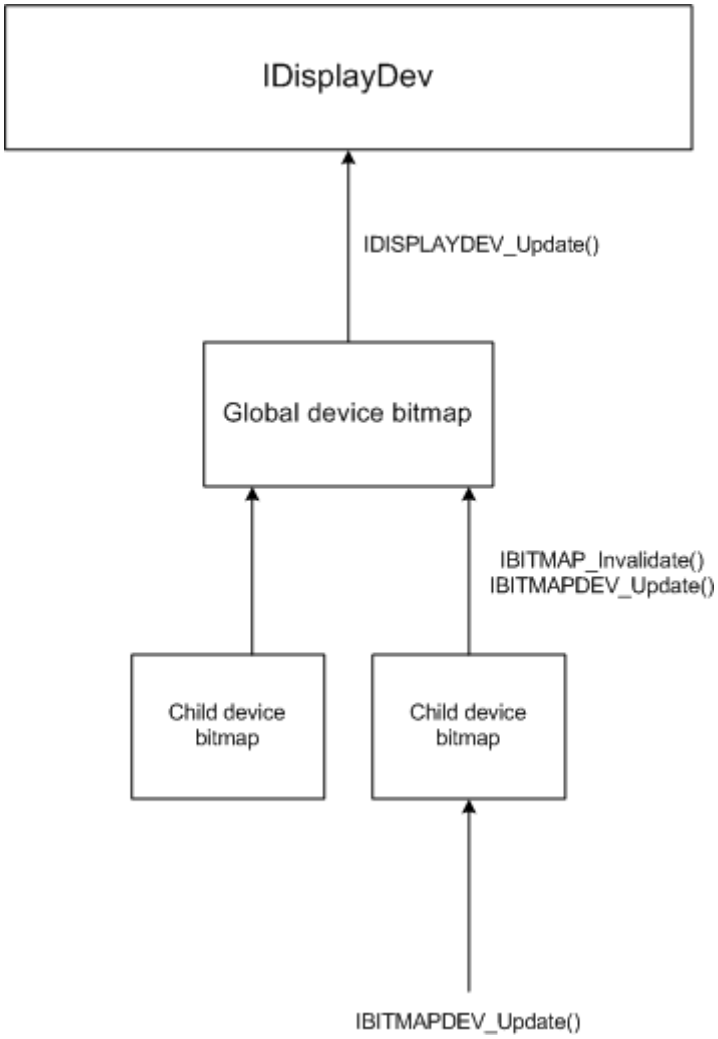
NOTE: Calling AEE_CreateInstanceSys() multiple times with AEECLSID_DEVBITMAP1 always returns the same IBitmap with the reference count incremented.

AEECLSID_DEVBITMAP*n*_CHILD creates a child of the corresponding global device bitmap. You should always create a new device bitmap for each AEECLSID_DEVBITMAP*n*_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_DISP*n* 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 Integrate the native UI in BREW enabled devices on page 64 for details and BREW UI: Integrating native apps with BREW on page 180 for more information on shim applications.

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.

IBacklight Interface

IBacklight interface enables applications to control various back lights available on a device. A device can have multiple components that support back light, such as primary display, secondary display, and key pad. Each device component supporting back light might support multiple back lights such as back lights of different colors. For example, the primary display might support red, green, blue, and yellow color back lights. To control a back light, applications create IBacklight interface by specifying the back light Class ID associated with the back light. IBacklight methods allow the application to control on/off status and brightness of a backlight.

Reference implementation of IBacklight (as specified in AEEBacklight.h) is provided in OEMBacklight.c.

Implementing 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.

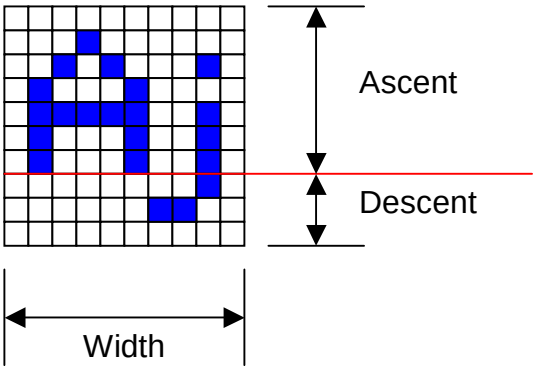
The IAppFrame interface is defined in OEMAppFrame.h. A sample implementation is provided in OEMAppFrame.c.

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 166).

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 ("|") are used as arbitrary character separators.

World のビジ
Shift-JIS: 0x57 | 0x6f | 0x72 | 0x6c | 0x64 | 0x82 0xcc | 0x83 0x72 | 0x83 0x64
BREW Hybrid: 0x5700 | 0x6f00 | 0x 7200 | 0x6c00 | 0x6400 | 0xcc82 | 0x7283 | 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 168 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 BREW UI Widgets. 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.

Enabling/disabling the annunciator using IAnnunciatorControl

The default OEMAppFrame.c, which is supplied in the Porting Kit now contains an implementation that handles enabling/disabling the annunciator bar through the IAnnunciatorControl interface.

To enable this interface OEMs must

1. Enable FEATURE_BREW_ANNUNCIATORCONTROL in OEMFeatures.h.
2. Modify implementation of OEMAnnunciatorControl.c to work the OEM's UI.

There are four class IDs that OEMAppFrame uses to determine which display's annunciator bar should be toggled:

AEECLSID_AnnunciatorControl_Display1
AEECLSID_AnnunciatorControl_Display2
AEECLSID_AnnunciatorControl_Display3
AEECLSID_AnnunciatorControl_Display4

The IAnnunciatorControl interface has two main functions: GetRegion and Enable. GetRegion returns the area of the screen that the annunciator bar occupies when it is enabled, and the Enable function is used to turn the region on and off as needed.

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.

Implementing Telephony support

BREW provides ITapi, ITelephone, and the ISMS family of interfaces (consisting of ISMS, ISMSMsg, ISMSStorage and ISMSNotifier) for the purposes of exposing the telephony and messaging feature of the handset to applications. For these interfaces to work properly, there is an OEM Layer component that must be implemented. This OEM Layer component is defined in OEMTAPI.h, OEMTelephone.h, OEMSMS.h and OEMSMSStorage.h. You must implement the functions declared in these header files to properly implement the Telephony support in BREW. See the *OEM API Reference Online Help* for details on the functions you must implement.

The BREW Porting Kit for MSM customers includes a reference implementation of the BREW Telephony OEM Layer that should interface directly with recent AMSS releases. You will see these reference implementations in the pk/src/msm directory of the Porting Kit installation.

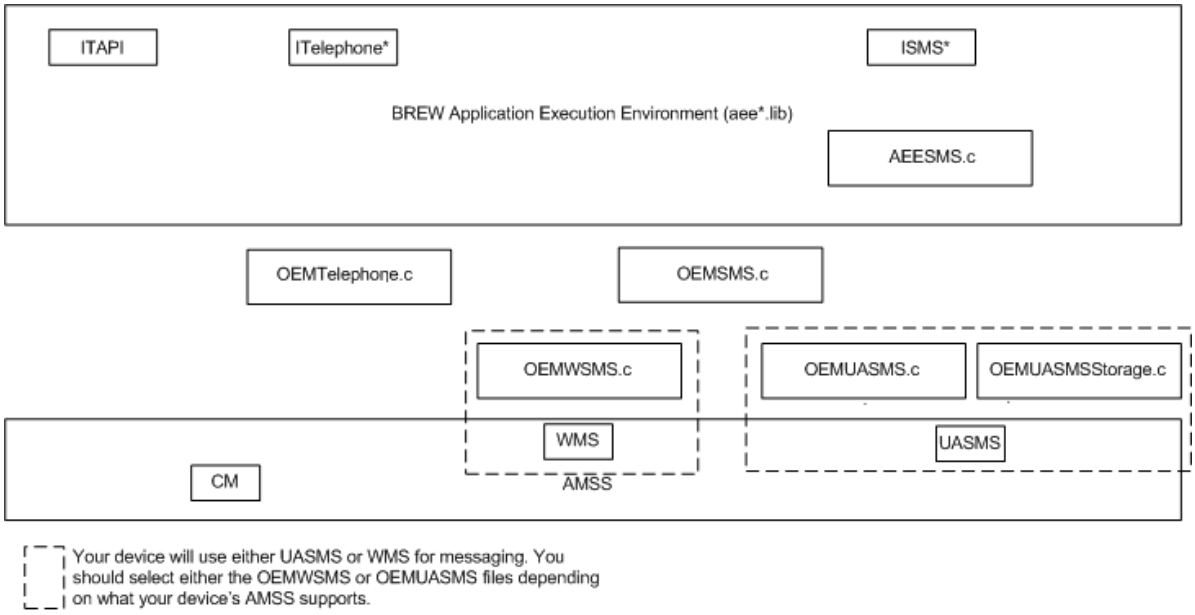
NOTE: For implementing ISMS support on recent AMSS releases, you can ignore OEMUASMS.c and OEMUASMSStorage.c. You should instead use OEMWSMS.c. OEMUASMS.c and OEMUASMSStorage.c are provided as a reference for OEMs who wish to port BREW onto those AMSS releases that use the UASMS module for SMS functionality. Most newer AMSS releases provide WMS for SMS functionality.

The figure below illustrates the relationship between the different files provided for BREW's Telephony support.

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 implementation regarding applications



Verifying implementation

See Run the PEK on page 74. You need to run the PEK for each of the appropriate modules to verify your implementation of the ITAPI, ITelephone, and ISMS interfaces. See the *BREW™ Porting Evaluation Kit User Guide* in the PEK documentation set.

This section is reserved for advanced topics on the BREW UI.

Call management

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

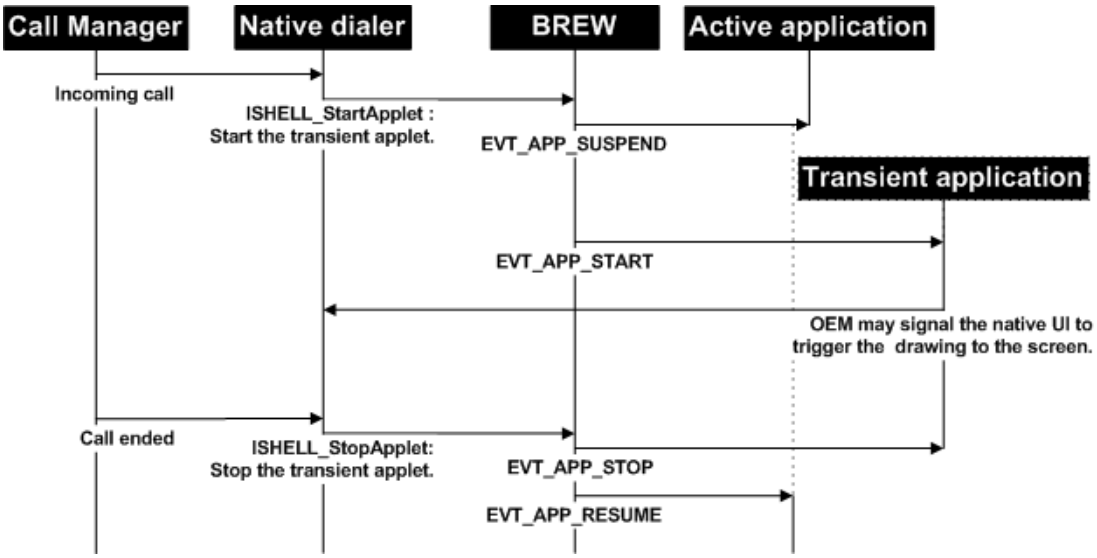
This section discusses the recommended mechanisms for handling device calls in conjunction with BREW. The recommendations are explained for the two different scenarios. In scenario (a), which shows a device with a non-BREW dialer, the dialer module is an external entity from BREW. See BREW-based UI or dialer on page 173. Scenario b shows a device with a UI dialer as a BREW application. See Managing call and position privacy on page 178.

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 BREW shim 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 BREW shim 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 173.

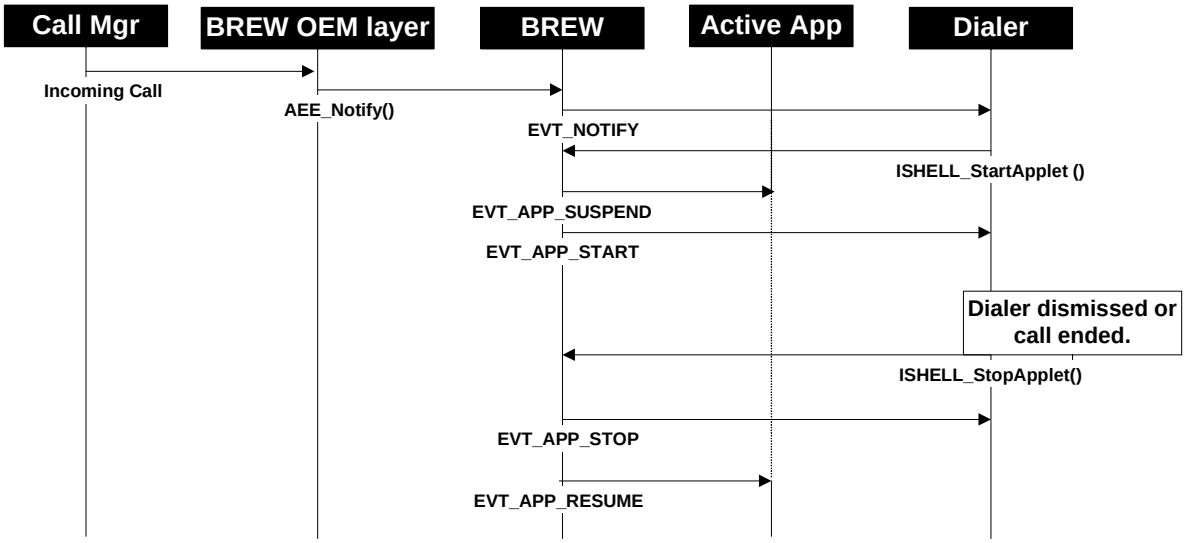
Call flow, scenario a: device with a non-BREW dialer



BREW-based UI or dialer

The BREW-based dialer applet, instead of the BREW shim 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.

Starting applications during voice calls

There are two types of BREW application launches to consider. The first of these is simply started by the OEM layer, where the OEM layer calls ISHELL_StartApplet() or a variant of this function in response to some external factor, such as a keypress or menu item selection. The second is an application self-start, where an application is registered for alarms, notifications, or events and calls ISHELL_StartApplet() or a variant on itself. BREW notifies the OEM layer to accept or reject any application starts. This notification is given in OEM_Notify() with the parameter OEMNTF_APP_START and the ClassID of the application that is requested to be started.

Step 1: Allow applications to be launching during a voice call

The general rule with the OEMNTF_APP_START notification is to allow you to prevent application launches when a critical task is occurring. Often, this critical task ambiguity has been interpreted to include when a voice call was in progress, so code may have been added to block application launches when a voice call was in progress. If this is the case in your system, you need to remove the code that prevents the application from launching during a voice call from the OEM_Notify() function with the OEMNTF_APP_START notification request parameter. This enables the application to start in BREW's environment. If your call application is a BREW applet, such as a BREW based UI, skip to step 3.

Step 2: Manage additional state switch cases

This opens the system up to more cases where the state may switch between native and BREW states if your call application is not a BREW application. It is recommended that you move your native UI into the shim design described in the Integrate native UI applications within BREW devices step on page 21, as it will ease state management burdens. Not doing this markedly increases the complexity of state management on your system. When a voice call is placed, whether from within or outside of BREW, a BREW shim application would be launched and act as a simple BREW applet placeholder.

NOTE: If your native call application still requires background native event feeds to track things like call time and other characteristics, you need to deviate from the default event dispatching behaviors covered in the Integrate native UI applications within BREW devices step on page 21. The discovery of the proper native handler, via CShim_GetEventHandler(), to invoke is done based on the active applet's ClassID.

If another application is started during the voice call, the discovery mechanism will not allow the correct native handler to be called. If the voice call application needs some events such as those described above for proper execution integrity, you need to add a flag when you are in a voice call and invoke the proper handler if the flag is set. Take special care that the events being passed to the native call application handler when this flag is set are benign native events and do not cause screen drawing or key handling functions to be invoked. These events should be thought of as processing events only and not as interactive events, much the same way that background BREW applications can still process callbacks in the background. The keys and screen now belong to the actively executing application, not the call application. A generic example of this alteration is shown in bold:

```
static void BREWEventHandler()
{
AEECLSID clsActive = ISHELL_ActiveApplet(AEE_GetShell());
…
// Near the end of the function send the native event
if( clsActive >= AEECLSID_FIRSTSHIM && clsActive <=AEECLSID_LASTSHIM
|| gpfnCallEvtHndlr ){
PFNUIEVENTHANDLER pfnEvtHndlr;
if( gpfnCallEvtHndlr ){
// This is set, so filter the events
if( ui_is_safe_background_event(ui_event) ){
gpfnCallEvtHndlr();
}
}else{
// Look up the handler based on the active class ID
if( (pfnEvtHndlr = CShim_GetEventHandler(clsActive)) == NULL ){
// error recovery and return
} else {
pfnEvtHndlr();
}
}
}
}
```

The assumptions given above are that the regular event processing for BREW applets is unaltered, and the applet that was launched during a voice call processes events and is completely unaware of the underlying call state. The gpfnCallEvtHndler is the call application event handler function pointer and is set when in a voice call and set to NULL when not in a voice call. The function ui_is_safe_background_event() is optional and provides a way to check whether the event is a background event such as a timer or inter-task communication vehicle that causes information processing or marshalling only and does not cause a user the call applet. But it should be treated like a background applet and should not be handling interactive events while still allowing background event-processing. It is crucial to guarantee that the safe event filter is effectively weeding out interactive-type events. The call application will also likely need modifications to be aware of when its own shim is not running as an added precaution. In your call application, you may need guarding blocks such as the following example illustrates:

```
AEECLSID clsActive = ISHELL_ActiveApplet(AEE_GetShell());
if( clsActive == AEECLSID_CALLSHIM ){
// Can handle user interactions
// Can access screen resources
}
```

Add these extra guards to prevent the code that draws to the display or handles user events from being mixed with benign processing code.

Step 3: Local-party-ended calls

Consult your carrier specification on how to handle local-party-ended calls. A possible scenario that may be required is that when the end key is pressed once the call is ended, a second press of the end key displays a dialog asking if the user would like to return to the original application that was running when the call came. Another possible scenario is that the user is directly returned to the call application (these both fit in Case A).

Case A

In the case that the user is returned to the call application or any other BREW application that was not currently executing, you need to intercept the end key event in the beginning of the BREW event handler function when gpfnCallEvtHndlr is set. Then, if the call applet is not active, call or start the applet with the same applet stack by dispatching the custom EVT_LAUNCH_APP on the call applet. For details on this custom event and why its use is necessary, see the Integrate native UI applications within BREW devices step on page 21.

```
AEECLSID clsActive = ISHELL_ActiveApplet(ps);
if( ui_event == END_KEY && gpfnCallEvtHndlr && clsActive !=
```

```
AEECLSID_CALLSHIM ){
ISHELL_PostEvent(AEE_GetShell(), AEECLSID_CALLSHIM, EVT_LAUNCH_APP, 0,
AEECLSID_CALLSHIM); // Effectively ISHELL_StartApplet()
return; // Do not translate for BREW to process at this time
}
```

BREW will now suspend the active application and resume the indicated applet. The example above uses the call applet, but you are free to launch any stored application that is needed. You may also respond to the end key by placing a dialog on the screen via a dialog shim or application and then processing the example above in the code block.

Another option available to use is a background applet that can respond to hooked key events. See the BREW SDK documentation for background applications and hooked key events.

Case B

In the case that the user is ending the call and not affecting applications, the end key still needs to be caught in the beginning of the BREW event handler and appropriate action taken. The call application shim should be closed also; you can do this by posting the event to have it close itself.

```
AEECLSID clsActive = ISHELL_ActiveApplet(ps);
if( ui_event == END_KEY && gpfnCallEvtHndlr && clsActive !=
AEECLSID_CALLSHIM ){
// Add code to end the voice call
ISHELL_PostEvent(ps, AEECLSID_CALLSHIM, EVT_CLOSE_SELF, 0, 0);
// Call gpfnCallEvtHndlr if you need to set some cleanup event
// Or cleanup code to execute such as annunciator cleanup
// and call log data recording
gpfnCallEvtHndlr = NULL;
return; // Do not translate for BREW to process at this time
}
```

Step 4: Optionally, add a UI to allow the user to launch BREW

This step depends on the carrier's requirements. If it needs to be fulfilled, check the specification for the target carrier on how this UI should look and behave and reference the Integrate native UI applications within BREW devices step on page 21 for implementation advice on launching BREW applications.

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.

See the OEM API Reference Online Help for more information

Enabling third party applications to send SMS messages to device inbox

Starting with this BREW release, BREW applications can now send SMS messages to the inbox of the main messaging application on the same phone. If an application wants to send an SMS message to the main messaging application's inbox, it can do so by specifying the ISMS message option MSGOPT_LOOPBACK_MSG as TRUE. These SMS messages are not sent to the network, but are routed back and can be handled by the main messaging applications. This code shows what an application needs to do while constructing an ISMS message to achieve this funcionality.

```
// Pseudo-code that allows third party applications to send the SMS
to the device Inbox as long
// as the OEM has added support for this in their OEM layer and the
device inbox.
smo[0].nId = MSGOPT_LOOPBACK_MSG;
smo[0].pVal = (void *)TRUE;
ISMSMSG_AddOpt(pme->m_pISMSMsg, smo);
```

Support for this has been added to the AEESMS header and library files. However, some changes need to be made by OEMs. These steps are required, depending on whether the device's main messaging application (that is responsible for displaying messages in the inbox) is written using the BREW ISMS family of interfaces.

BREW ISMS-based main messaging applications

If the main messaging application is completely based on the ISMS interfaces:

1. The loopback messages can be retrieved by registering for the mask, NMASK_SMS_TYPE_LOOPBACK.
2. The incoming message contains a new message option, MSGOPT_LOOPBACK_FROM_APPLET, which contains the AEECLSID of the origination applet. This can be used by the main messaging client to trace back the sender of the message.
3. No changes to the OEM layer are needed.
4. OEMs can look into OATSMS for tips on implementation where a new PEK test SRLOOPBACK_NoPayload has been added.

   The test sends a loopback message and ensures it is received by OATSMS. It specifically tests the message options MSGOPT_LOOPBACK_MSG and MSGOPT_LOOPBACK_FROM_APPLET. OAT is registered to accept NMASK_SMS_TYPE_LOOPBACK to receive these loopback messages.

AMSS/WMS-based main messaging applications

NOTE: WMS stands for Wireless Messaging Service. See wms.h in the AMSS engine layer for more information.

On device builds, which have the main messaging application built directly on top of the wms.h API, OEMs need to incorporate support for this feature in the OEM layer.

The function, OEMSMSCust_LoopbackMsg(), is a placeholder that will be invoked by the SMS implementation for these loopback messages, allowing OEMs to direct the messages to the inbox of their main messaging application. The contents of the SMS message are contained in SMSCacheEntry structure. Refer to OEMSMS_priv.h for details about the fields of SMSCacheEntry. This stub function is defined in a new file, /pk/src/msm/oem/OEMSMSCust.c, which the OEM can use to fill in the implementation.

BREW UI: Integrating native apps with BREW

BREW and native applications

This section describes the steps to be followed by OEMs to support the coexistence of BREW applications and non-BREW applications, also referred to as native applications, on a device. The purpose of this procedure is to make it seamless for a user to switch from one application to the other, regardless of whether the application is a BREW application or a native application, using a shim application. The purpose of this shim application is to ensure that drawing to the screen is done in the context of a BREW application. For example, the user can switch from a game (BREW-based) to a browser (native application) and be able to return to the same instance of the same game when finished using the browser.

Following is a scenario that you can avoid if the instructions in this section are followed:

1. The user launches a BREW application (game).
2. The user presses a hot key to launch the browser (native application).
3. The user presses a hot key to launch the AppManager (a BREW application).
4. Using the AppManager, the user tries 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.

If OEMs follow the guidelines in this section, the same instance of the game can be resumed, which will enhance the user experience. One of the reasons for the scenario described above is that BREW is explicitly suspended and resumed by using AEE_Suspend() and AEE_Resume(). Following is a method to avoid invoking these functions.

Whenever the native software layer takes control of a display, a corresponding BREW shim application is started.

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. The following guidelines can be used in scenarios such as handling incoming calls and incoming SMS when a BREW application is running so that 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. See BREW UI: Call handling on page 172 for more information on call handling.

Call handling scenarios

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) is running, the device receives an incoming call.
2. AEE_Suspend() is called from the native software layer.
3. A corresponding UI dialog is put up, advising that the user is in the call.
4. The call ends.
5. AEE_Resume() is 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 BREW shim application starts by calling ISHELL_StartApplet().
3. HelloWorld is automatically suspended.
4. The BREW shim 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 BREW shim application is closed by calling ISHELL_CloseApplet().

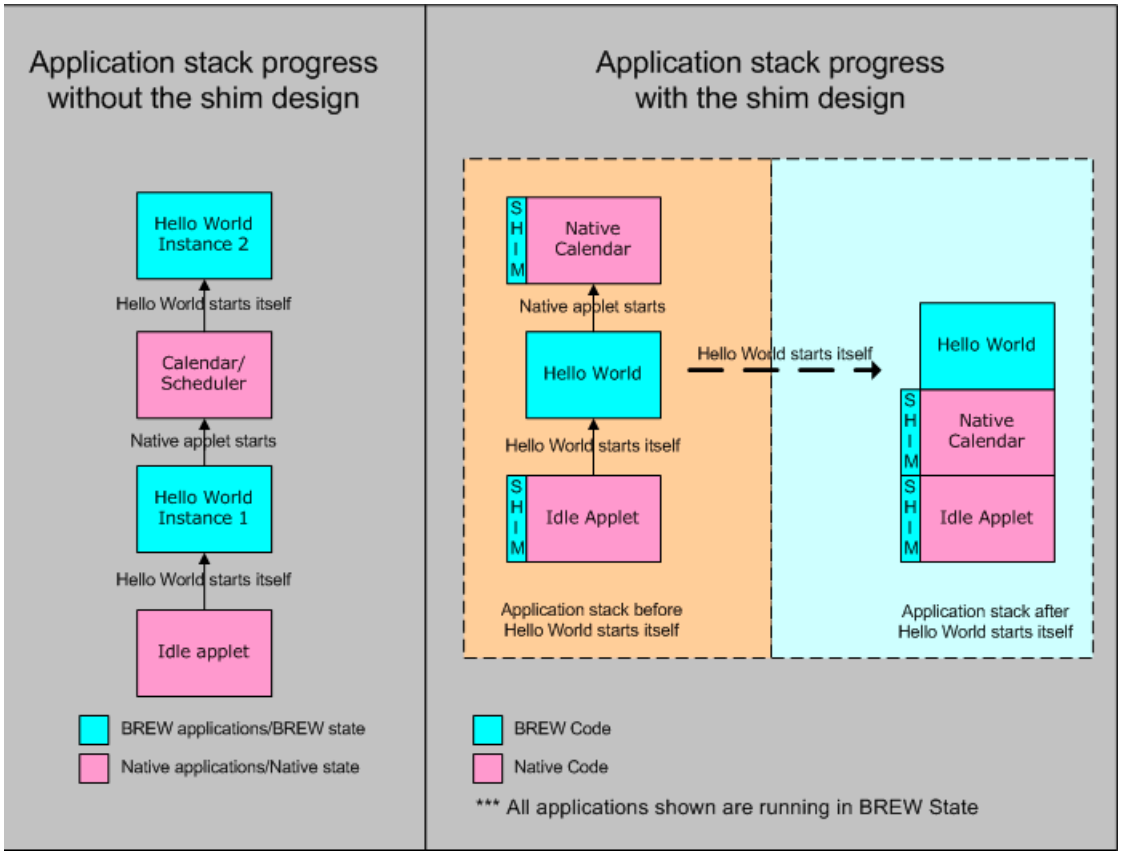
7. BREW automatically resumes the application (HelloWorld) that was running when the call was received.

As shown in the example above, OEMs need not invoke AEE_Suspend() or AEE_Resume() directly. This is taken care of 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 BREW shim application, which the OEM 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



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 an SMS message or alarm. For example, if step 2 on page 181 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 integrate native UI applications in a BREW device

NOTE: See the samples provided in the Porting Kit example code. There are comments in the code labeled “TODO” where OEMs should consider making changes.

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 stateapp.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 stateapp.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
#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 this section.

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 stateapp.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 stateapp.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 stateapp.h header or the header in which you defined your ClassID list:

```
#include stateapp.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 stateapp.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 ] );
}
}
```

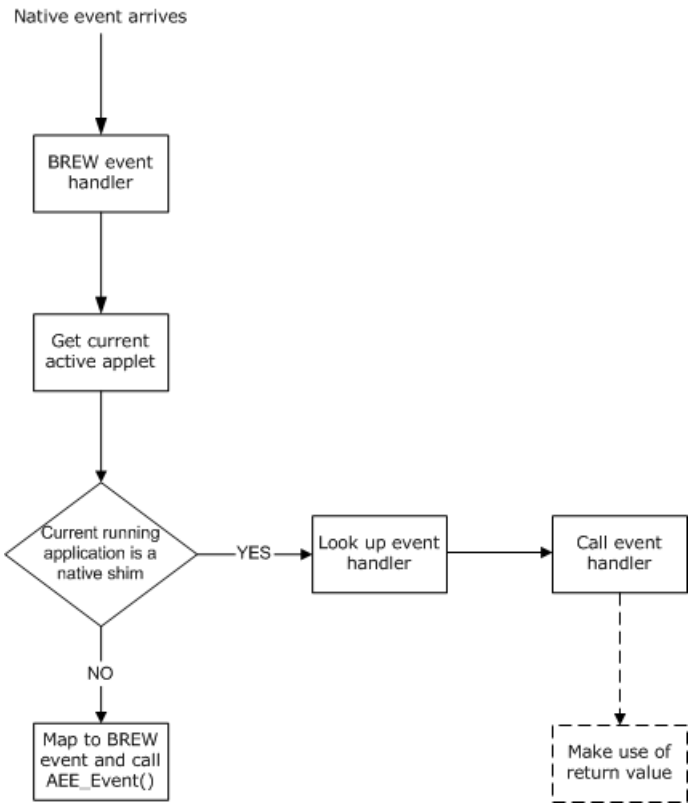
This process verifies if the current state is the BREW state requesting to be popped. If it is, a further check is made to see if the pop was triggered by a shimmed application. This method ensures that the BREW state can be popped when needed and closes the shimmed applet when needed as well.

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.



When an event comes into 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 stateapp.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

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 stateapp.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 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.

Verification of the 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.

Changing a language from a native UI menu

In BREW 3.1, when the language is changed from a native UI menu, BREW continues to display language resources in the old language until the phone is power-cycled. BREW does not call OEM_GetDeviceInfo() everytime it loads a resource. Instead, it internally maintains the state about the device information. What BREW requires is a way to let it know when any of the device information struct members (including the current language) change by issuing a callback, so that BREW's internal device information stucture does not become stale.

To let BREW know that there has been a language change

1. In your native UI, when the language is changed, send a callback to BREW:

```
#include "AEEShell.h"
#include "AEE_OEMDispatch.h"
AEE_IssueSystemCallback(AEE_SCB_DEVICE_INFO_CHANGED);
```

   BREW will mark its internal deviceinfo structure as stale, and will call OEM_GetDeviceInfo() the next time you load the resource. The language will then be changed.

Managing Resources

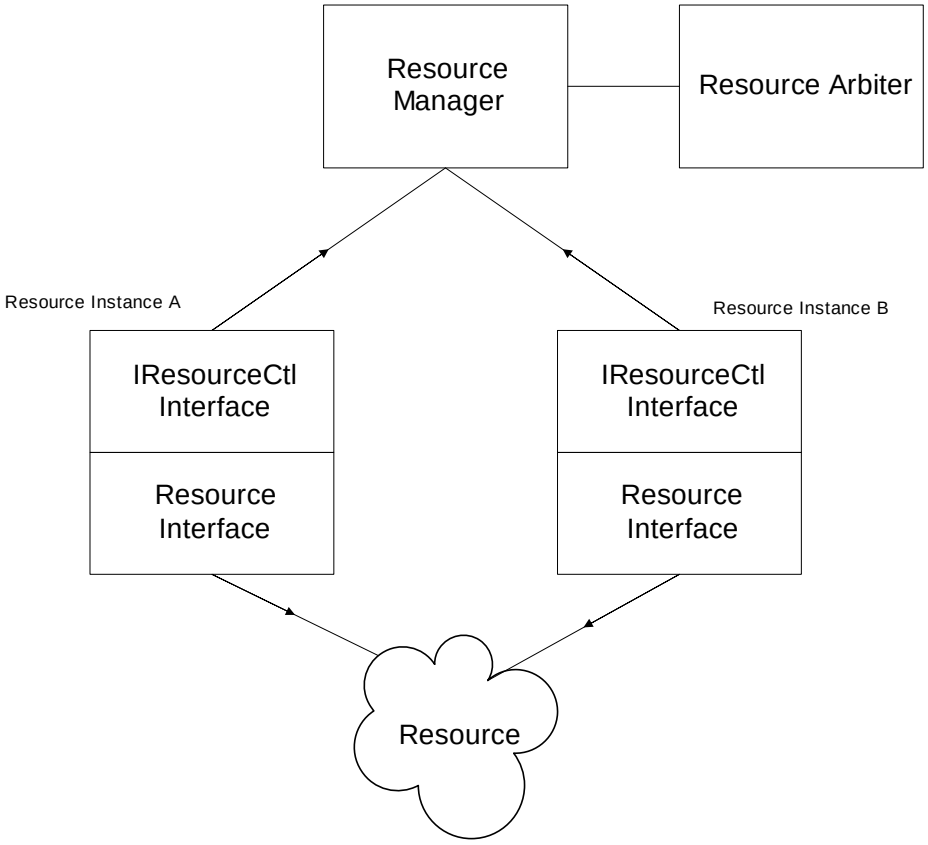
This section describes the BREW resource management mechanism. BREW currently implements top visible and ISound with resource management. Resource managment provides a generic means for objects, including BREW applications, to control resource access. The resource manager also coordinates and manages 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

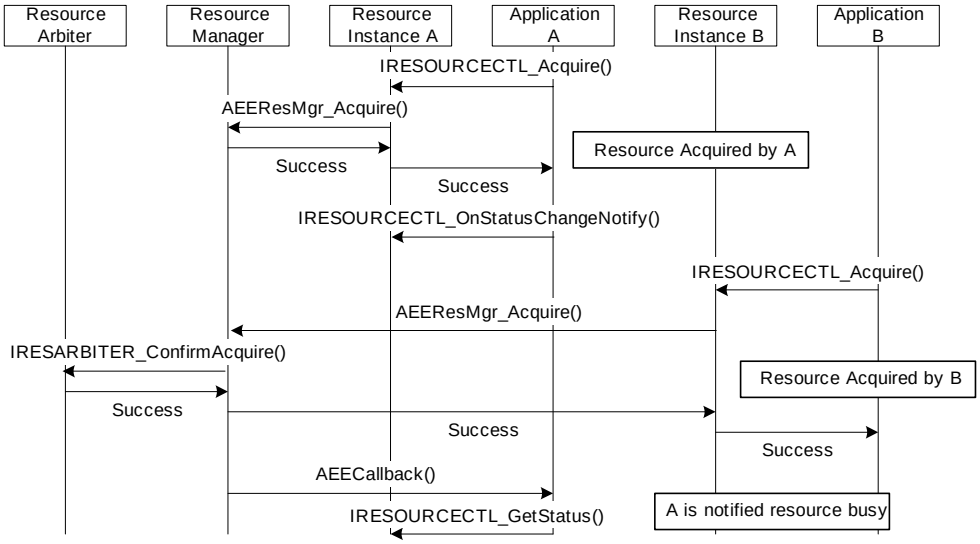


When you create an instance of the resource interface, it includes the IResourceCtl instance. The IResourceCtl instance interacts with the resource manager to acquire and free the underlying resource.

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);
}
```

BREW UI: App stacking and app history

This section describes the BREW Application History feature and how this feature gives OEMs the capability of managing the UI and running applications on their devices with BREW UI or BREW Application/Extension solutions.

BREW 3.1 introduces a new feature in BREW architecture called Application History, and it exposes the IAppHistory API to be used by BREW applications. The Application History feature allows BREW to maintain separate application stacking information, which can be modified by applications using the IAppHistory API. This ability to modify the application stack using Application History gives the OEMs the capability of managing UI and running applications on their devices with BREW UI or BREW Application/Extension solutions.

Whenever there is an attempt to start a BREW application and make it top-visible, for example, when ISHELL_StartApplet() or one of the macros based on this function is invoked, BREW will attempt to create an Application History entry for this application. If the application does not already have a history entry, for example, if it is not already in the list of suspended applications, a new history entry will be created.

If this application is already in the history list, the following rules will be used to decide whether to create a new entry for this application, or to move the already existing history entry to the top of the list:

- If the application set resume data previously (when it got suspended) by invoking IAPPHISTORY_SetResumeData(), then a new history entry will be created for the application when ISHELL_StartApplet() is invoked again.
- If the application did not set resume data previously by calling IAPPHISTORY_SetResumeData(), then a new history entry will not be created for this application when it starts again. Instead, the already existing history entry will be moved to the top of the list (to make this application top-visible). This is done to maintain backward compatibility on applications written to BREW versions prior to 3.x.

Calling IAPPHISTORY_Remove() on the last entry associated with a loaded applet will close the underlying applet. In this case, if the applet that needs to be closed is currently running, it will be closed asynchronously. For example, calling IAPPHISTORY_Remove() will not result in synchronously sending EVT_APP_STOP to the application if the applicaiton is currently running. That will be done asynchronously.

Calling IAPPHISTORY_Remove() on the top-visible applet will cause the application associated with the next history entry to be resumed.

Other than using the IAPPHISTORY_Remove() API to remove a history entry, ISHELL_CloseApplet() will also remove the history entry (if one exists) for the caller. Calling IAPPLETCTL_Stop() will not remove a history entry unless it is the top-visible application.

API's like IAPPHISTORY_Top(), IAPPHISTORY_Bottom(), IAPPHISTORY_Back(), IAPPHISTORY_Forward() can be used to traverse this Application History stack. The API's, IAPPHISTORY_Insert(), and IAPPHISTORY_Move(), can be used to add and move entries within the Application History stack.

A call to AEE_Suspend() to suspend BREW will cause a dummy application history entry to be put at the top of the Application History stack. The application class ID for such a dummy entry is NULL. When AEE_Resume() is called, BREW removes this dummy suspend entry from the Application History stack and resumes the underlying application in the Application History. If there are no applications to be resumed, BREW sends an OEMNTF_IDLE notification to the OEM Layer. Two or more consecutive calls to AEE_Suspend() will result in just a single dummy application history entry. It will not result in multiple dummy history entries.

Appendix A: 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.
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.
DMI	Diagnostic Monitor Interface	A QCT (AMSS) 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.
HWID	Hardware Identifier	The HWID is the factory assigned serial number(s). Depending on the device, it may be an ESN, IMEI, and/or an MEID. The primary HWID is the ID used by BREW to identify the device. The primary ID can be any one of the HWIDs supported as long as it remains constant for the life of the device.
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	
MCF	Media Content File	
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.

pESN	Pseudo-ESN	The pseudo-ESN is computed ESN, using 0x80 as its Manufacturer Code, followed by a 24 bit hash of the 56 bit MEID. It replaces the ME ESN for MEID-equipped terminals.
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	
TCB	Task Control Block	
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	

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 AMSS. 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 206for 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 206.
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 206 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.

The following commands must be implemented on the mobile device for it to be DMI-compliant.

DMI compliance command

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 AMSS Status Request/Response	The status message asks for current AMSS status information
15	N/A	Logging Mask Request/Response	The status message asks for current AMSS status information.
16	N/A	Log Request/Response	Retrieves a single queued log item from the AMSS. The AMSS 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/AMSS packet interface in use by the AMSS by sending a Diag Version Request Message. The AMSS responds by sending a Diag Version Response Message containing the version number. If the version used by the AMSS 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 AMSS.

Command code	Operation code	Packet name	Description
31	N/A	Message Request/Response	Retrieves the AMSS diagnostic message. As a diagnostic aid, the AMSS records text messages at various points in the execution. The messages provide developers with some insight into the behavior of the AMSS program. Message level (minimum message severity level): • 0000: all messages (MSG_LVL_LOW) • 0001: medium and above (MSG_LVL_MED) • 0002: high and above (MSG_LVL_HIGH) • 0003: error and above (MSG_LVL_ERROR) • 0004: fatal error only (MSG_LVL_FATAL) • 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 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 AMSS locks the device, which prevents user input, and processes the Device Emulation Keypress Messages. On invoking this Unlock request, the AMSS unlocks the handset.

Command code	Operation code	Packet name	Description
41	Mode = 1	Mode Change Request/Response Message	Puts the AMSS (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: • Memory Peek Request/ Response • Memory Poke Request Response • NV Memory Peek Request/ Response • 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 AMSS (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 AMSS.
93	N/A	Extended Logging Mask Request/Response	The DM sends the Extended Logging Mask Request Message to the AMSS 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

E

F

G

H

I

J

K

T

U

V