Programming Model for Developers

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

This documentation was written for use with Brew Mobile Platform, software version 1.0.2. This document and the Brew Mobile Platform software described in it are copyrighted, with all rights reserved. This document and the Brew Mobile Platform software may not be copied, except as otherwise provided in your software license or as expressly permitted in writing by QUALCOMM Incorporated.

This technical data may be subject to U.S. and international export, re-export or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Sample Code Disclaimer:
This QUALCOMM Sample Code Disclaimer applies to the sample code of QUALCOMM Incorporated (“QUALCOMM”) to which it is attached or in which it is integrated (“Sample Code”). Qualcomm is a trademark of, and may not be used without express written permission of, QUALCOMM.

Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, QUALCOMM provides the Sample Code on an "as is" basis, without warranties or conditions of any kind, either express or implied, including, without limitation, any warranties or conditions of title, non-infringement, merchantability, or fitness for a particular purpose. You are solely responsible for determining the appropriateness of using the Sample Code and assume any risks associated therewith. PLEASE BE ADVISED THAT QUALCOMM WILL NOT SUPPORT THE SAMPLE CODE OR TROUBLESHOOT ANY ISSUES THAT MAY ARISE WITH IT.

Limitation of Liability. In no event shall QUALCOMM be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of the Sample Code even if advised of the possibility of such damage.

HT80-VT500-127 Rev C
August 25, 2010

Contents

Programming Model for Developers

This document provides a discussion of the Brew® Mobile Platform (Brew MP) programming model.

For introductions to development environments and setup specific to languages, see the Brew MP Primers on the Brew MP website. This document is meant to provide a more in-depth exploration of programming for the Brew MP platform, to bridge the Brew MP Primers and Technology Guides.

Key aspects of the Brew MP programming model include:

How Brew MP relates to BREW

BREW has traditionally been positioned as a bundled end-to-end offering combining an application platform, SDK, and application distribution system. Operators deploying the BREW Distribution System (BDS) leveraged applications targeted for distribution by the BDS. Along with BREW, Qualcomm has also released a number of related products including uiOne™ SDK, uiOne HDK, and BREW UI Widgets.

In contrast, Brew MP is focused on expanding the capabilities of the device as an open application and service platform. Brew MP provides a broader and deeper set of APIs spanning all layers of the device. While combining elements of the traditional BREW, uiOne and Widgets offerings, Brew MP provides substantial additional functionality including support for OS functionality, Flash, and window management.

Please note that the user mode support mentioned throughout this document will not be fully supported until Brew MP 1.1+.

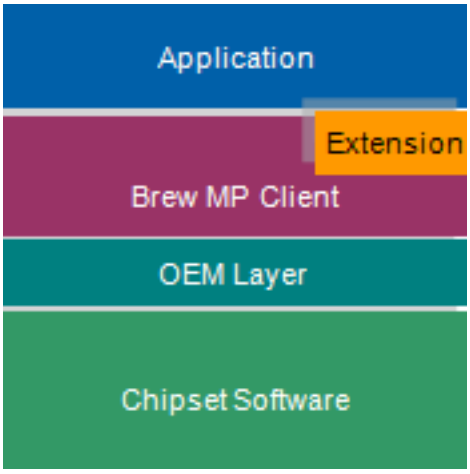
For information on how the Brew MP SDK Tools relate to the BREW Tools, see *Tools for Veteran Developers, Tools for Newbies* on the Brew MP Developer Network (Brew MP Dev Net).

Applications and extensions

A Brew MP application is a self-contained software package that exposes at least one applet class (implements IApplet interface) that can be loaded and executed in the BREW Shell (or thread).

A Brew MP extension is a self-contained software package that exposes one or more non-applet classes with interfaces that can be accessed by any number of Brew MP applications for extended functionality. Extensions are similar in concept to a software plugin for a PC application. See in-process classses on page 7 for more information.

Relationship between an application and extension



The main difference between Brew MP extensions and applications is that classes in an extension must expose their virtual function tables to the other external classes. Classes in extensions can be written without a UI, since the UI of the calling application can be used.

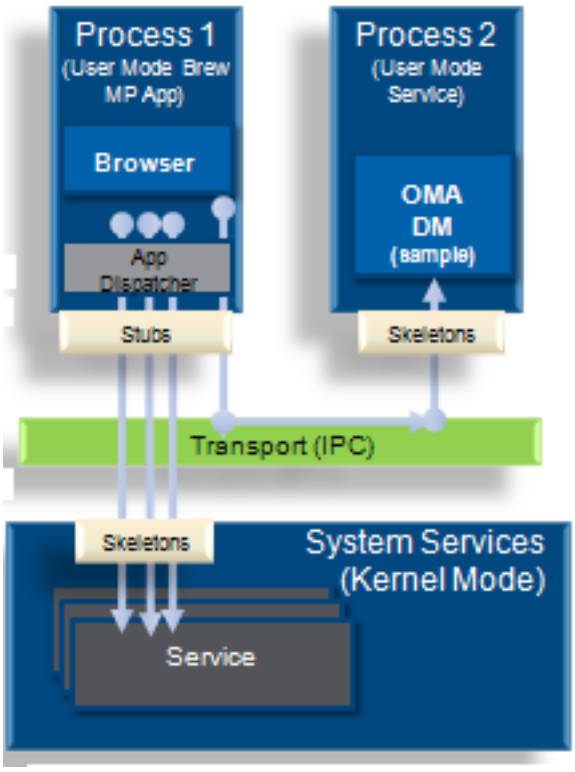
An application can create an instance of a class in the extension by calling IShell_CreateInstance() or IEnv_CreateInstance(). Then, each function of the class is accessible though the macros defined in the header file.

Applications are hosted inside specialized application processes and are controlled by an application dispatcher mechanism that governs the application life cycle, which includes the following:

- Startup
- Event & notification processing
- Suspend/resume
- Shutdown

Extensions are defined as software packages or modules that contain non-applet classes, which means that those classes can be in-process classes or service classes. Both in-process and service classes expose public APIs (interfaces). An in-process class can be thought of as code extension of the caller; a service class can be thought of as a background system process providing access to unique functionality.

Brew MP exposes most of its own functionality by way of system services. Leveraging the stubs and skeletons transport model, service classes can either run in the kernel process or as isolated user mode processes, as shown in the following diagram. See User mode on page for more information.



For more information on services, in-process classes, and service classes, see Classes on page 7.

Brew MP architecture

Brew MP has four core layers:

- OS Services - abstracts kernel and memory management, provides component management, process, and security across the platform. The OS Services layer provides portability to both Qualcomm and non-Qualcomm chipsets.
- Platform Services - includes modem, multimedia, and general service features, and is also the layer where Brew MP application APIs reside.
- Application Environment - provides the foundation for applications running on Brew MP. Supports application services such as Flash, Lua, TrigML™ , Widgets (BUIW), and window manager.
- Applications - can be developed in C/C++, Flash, TrigML , and Java.

The Brew MP application model is event-driven

Brew MP applications respond to events sent from the operating system. Brew MP applications do not contain a main program loop; all input is received through events. This model provides for clean, efficient execution with minimum demand on system resources and simple, task-based development.

Brew MP leverages an object-oriented modular design that governs system architecture, security, and access to all APIs and services. This design provides dynamic platform extensibility across language environments (C/C++, Flash, etc.). The key components of Brew MP application architecture are the following:

- Interfaces on page 6

- classes on page 7
- modules on page 17

Qualcomm Component Model (QCM)

The Qualcomm Component Model (QCM) is a programming model in which software is built as components.

QCM provides the infrastructure for Brew MP to extend the capabilities of the platform by adding new services and allow those services to be dynamically discovered and used. BREW, Brew MP, and OS services are all QCM compliant.

The QCM:

- Establishes a contract and specification between providers of services and their users.
- Separates the specification and implementation of the services. The specification describes the functionality the software service, or implementation, provides.
- Enables the users and the providers of the services to undergo changes without breaking each other.
- Enables services to be dynamically discovered and created.

Users of the service must conform to this specification to access the functionality, and only need to know about the specifications of the services rather than how the services are actually implemented.

QCM is not a platform. BREW and Brew MP are the platforms that leverage this programming model to have their APIs built as components. This programming model consists of the following:

- interfaces on page 6
- classes on page 7
- components on page 17
- modules on page 17

Interfaces

An interface is a software contract and specification between an implementing class and its using client. Interfaces provide the definition of a particular grouping of APIs in a functional object.

Interfaces are identified by unique 32-bit AEEIIDs, included in the interface definition. For public interfaces, the interface ID should be obtained using the Brew ClassID Generator. The following is an example of the interface ID definition.

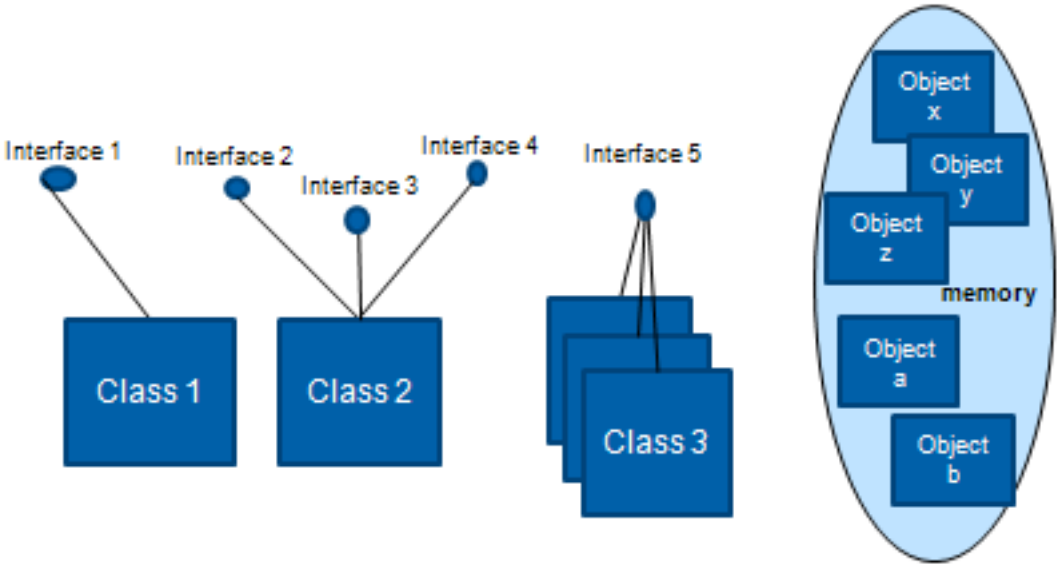
```
const AEEIID AEEIID_IFoo = 0x00000000; /* not a real IID */
interface IFoo : IQI
{
    /* interface body */
};
```

Brew MP enforces strict rules for interface construction, naming, and life cycle, which ensure platform compatibility and security. APIs are exposed by modules as objects associated with interfaces and classes. See classes on page 7 for more information.

There are two kinds of interfaces:

- Interfaces that use dynamic binding: true run-time interfaces that conform to QCM. These interfaces are commonly referred to as QCM interfaces.
- Static APIs: conventional C APIs resolved during the link step of the build.

An interface describes how to interact with the instances of the class. The following diagram illustrates the relationship between interfaces and classes, and how objects manifest in memory. The classes and interfaces on the left are defined in the code for the applet; the objects in memory shown on the right are the instantiations of classes that are stored in memory when the applet is executing.



Defining interfaces in IDL

A key feature of Brew MP is support for interfaces across languages and environments. For example, a developer can access many of the same APIs in C/C++ and Lua. This abstraction is independent of the language used to implement the underlying component. A component may be implemented in Lua and accessed from C/C++, or vice versa.

Brew MP provides an Interface Definition Language (IDL) capability to allow developers to create high-level specifications for interfaces that can then be mapped to many other languages.

The IDL mechanism does the following:

- Describes interfaces in a clean and concise manner.
- Automates correct header-file generation across languages.
- Enforces rules to simplify development of inter-process code. Brew MP also provides a compiler to auto-generate proxies.
- Enables interpreted environments by way of auto-generated language-specific proxies, avoiding the need to define and implement protocols for each area of functionality.

Brew MP's IDL support is based on OMG (CORBA) IDL with some specific omissions and additions to support Brew MP. The IDL mechanism currently supports C/C++ and Lua, with support for ActionScript under development. For more information on IDL, see the *QIDL Reference*, and the QIDL Compiler section of the Brew MP *Tools Reference*.

Classes

In Brew MP, software programs are written as classes. A class is a user-defined type that encapsulates data and behavior (i.e. functions) to provide implementation of one or more interfaces it exposes.

Classes are identified by unique 32-bit AEECLSIDs. The AEECLSIDs supported for a module are specified in the module's Module Information File (MIF). When a class is instantiated, it becomes an

object, which is an instance of the class in memory that maintains the data members of the class and the VTable to the code of all its supported methods.

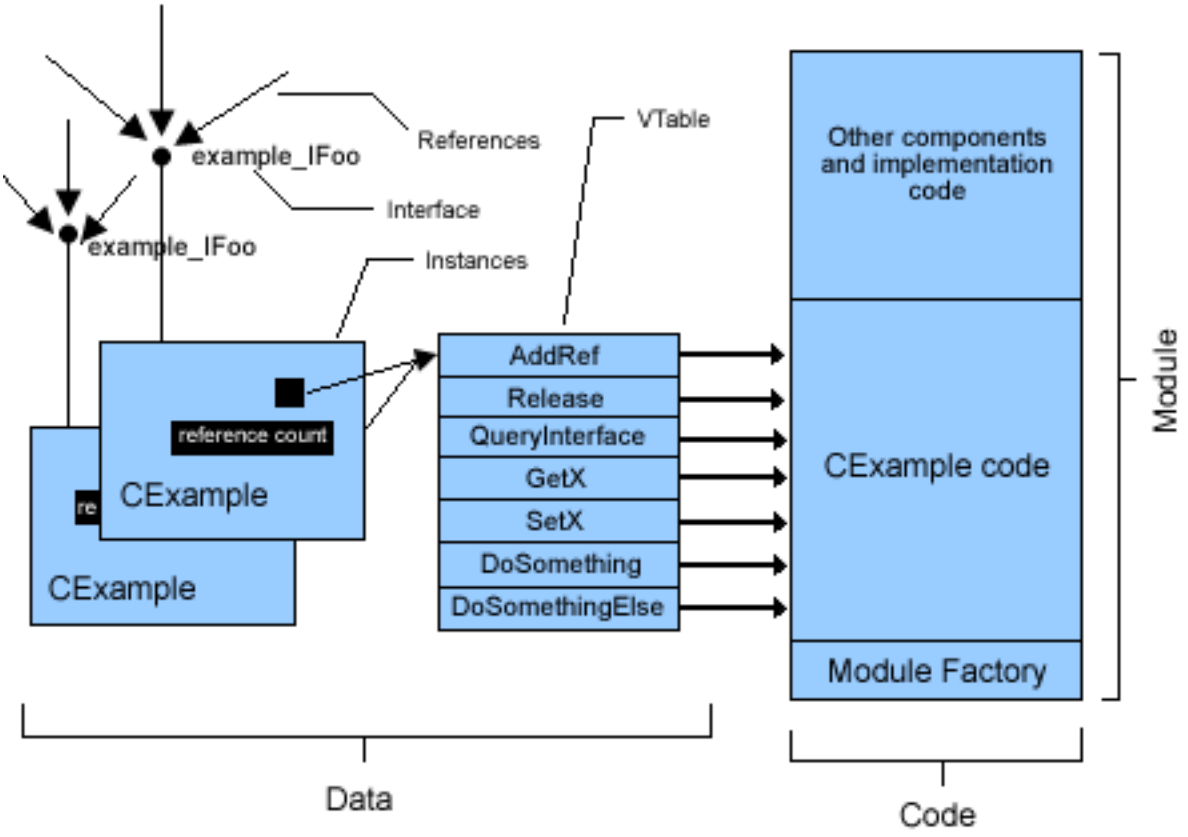
C/C++ programs in Brew MP implement three types of Brew MP classes:

- applet classes
- in-process classes
- service classes

The table below describes the module formats and types, and wizards for these classes.

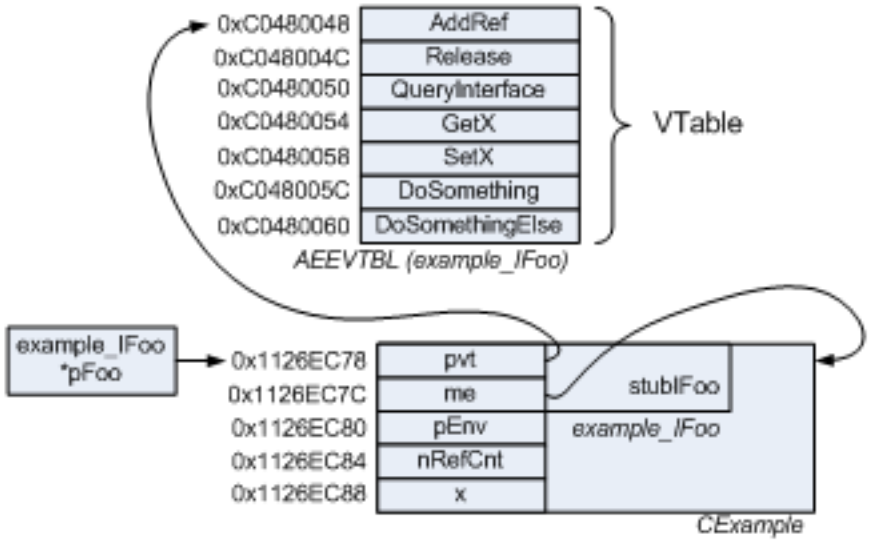
Brew MP class types	Supporting module formats	Applicable Brew MP module types	Supporting IDE Wizards
Applet class	MOD or MOD1	Application	Application (applet class)
In-Process class	MOD or MOD1	Application or extension	Extension (in-process class)
Service Class	MOD1	Application or extension	n/a

In the example below, the interface is called example_IFoo, and the instantiated class (object) is called CExample. As shown below, there can be multiple instances of the same class. Every CExample object has a pointer to the vtable and a reference count that keeps track of the number of references to the interface.



A component can have multiple interfaces, each of which provides a different functionality set for clients with different roles. Multiple interfaces allow multiple references to multiple vtables, each of which provides different functionality. All references to the same interface point to the same vtable, which is in the same area of memory, and is therefore the same code.

Since clients receive and operate on a reference (the address of a pointer to the interface), developers can dynamically cast that pointer type. It is common practice to have a "smart" pointer to the actual instance. In the picture below, example_IFoo creates an instance of CExample and locally stores pvt (the pointer to the Vtable). The second field stores me, which points to the actual instance of CExample.



Typically, a class can contain one or more interfaces. See interfaces on page 6 for more information.

There are two commonly used mechanisms for creating instances of a class with QCM interfaces.

1. Using the component infrastructure. Classes with QCM interfaces may register (or advertise) with the component infrastructure. These classes are typically identified by 32 bit unique identifiers referred as ClassIDs. Instantiation of these classes is done using IEnv_CreateInstance(), or IShell_CreateInstance() if the class has access to the IShell object. In the following example, AEECLSID_CallMgr represents a ClassID of the call manager class.

   ```
   IEnv_CreateInstance(piEnv, AEECLSID_CallMgr, (void**)&piCallMgr)
   // returns an instance of call manager class in piCallMgr
   ```

2. Using a factory class. These classes have interfaces that output instances of another class. A factory class is typically used to express the additional initialization parameters to make an object. The following example returns an instance of class Call, initialized to its originating state with the listener and destination phone number.

   ```
   ICallMgr_OriginateVoice(piCallMgr, "8585555555", piListener,
   &piCall)
   ```

Classes with static APIs are instantiated by directly invoking their constructor.

Types of classes in Brew MP

There are three types of classes in Brew MP:

- applet class on page 11
- in-process class on page 11
- service class on page 12

In-process and service classes are non-applet classes, and have some similar behaviors. Non-applet classes are instantiated using a unique ClassID via IEnv_CreateInstance(), or IShell_CreateInstance() if the caller has access to IShell. Non-applet classes are released via the Release() method exposed by the classes. When a non-applet class is instantiated, the default interface is returned to the caller and the caller can use the QueryInterface() method exposed by the class to discover other interfaces supported by the class.

Applet class

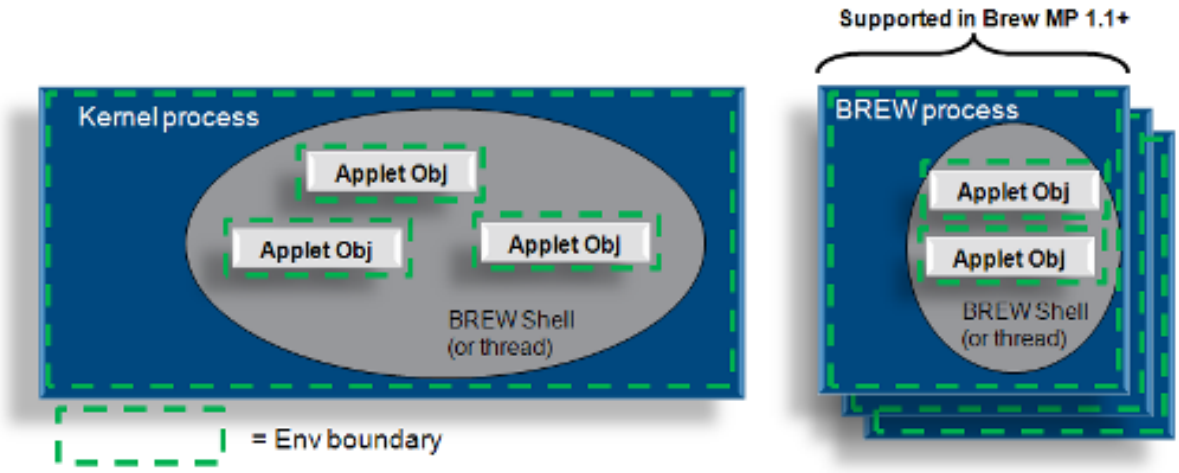
Applet classes implement IApplet and are identified, dynamically discovered, and instantiated using a unique Applet ID (see Unique ID's on page 60). Applet classes are also known as applets or applications in Brew MP. These are IShell-dependent classes; they can only be instantiated inside BREW Shell. IShell_StartApplet() or related APIs are used to start an applet and IShell_CloseApplet() or related APIs are used to delete or terminate a running applet. The applet class is declared via the Applet primitive in the CIF. See IShell on page 50 and MIF and CIF on page 64 for more information.

The following is an example of declaring an applet in CIF:

```
Applet {
  appletid = AEECLSID_MyApplet,
  resbaseid = 20,
  applethostid = 0,
  privs = { AEEPRIVID_UDP_NET_URGENT, AEEPRIVID_FS_FULL_READ },
  type = 0,
  flags = 0,
  newfunc = MyApplet_New,
}
```

MyApplet_New is the constructor of the applet class written in C/C++ code, and is invoked when the applet is started by IShell_StartApplet() using AEECLSID_MyApplet. applethostid = 0 indicates that the applet class is started in the kernel process. newfunc is explicitly specified in the CIF for MOD1 files. For MOD files, the constructor is set up by helper files such as AEEModGen.c.

For more information on CIF, see the *Resource File and Markup Reference*. For information on Env, Environments on page 18.



In-process class

In-process classes are non-applet classes that service the caller's request in the caller's process. Most BREW APIs are implemented as in-process classes. These classes use the permissions and quota limits of the caller to access resources. In Brew MP, all classes with static APIs are in-process classes with respect to the user. In-process classes are usually contained in extensions in Brew MP, and can be thought of as code extensions for the caller. The in-process object is created inside the Env of the caller and its methods invoked by the caller result in direct function calls. It shares the same privileges as the caller, and if created as a singleton, there exists only one instance of the class in the Env of the caller.

See extensions on page 3 for more information on extensions. See Environments on page 18 for more information on Env.

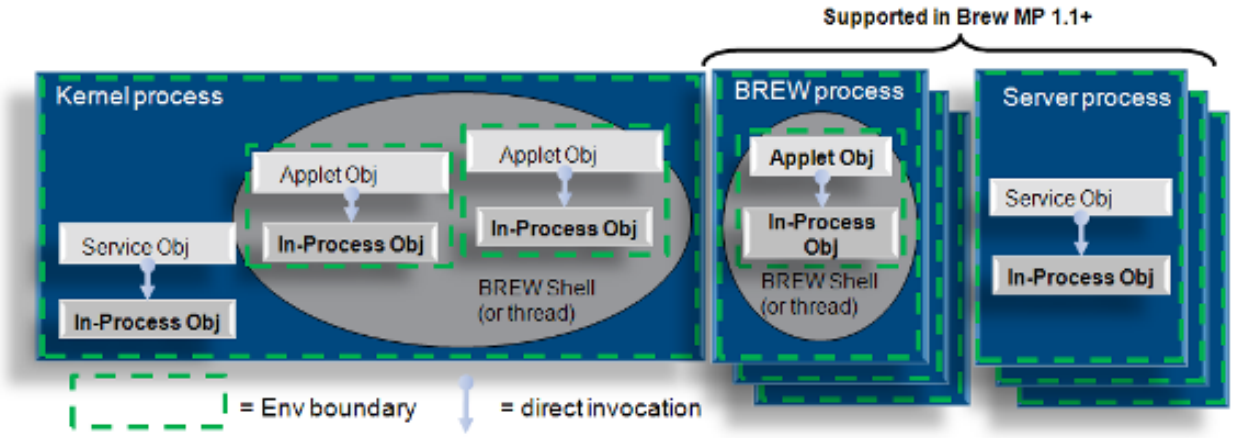
In-process classes that provide QCM interfaces can be registered with the component infrastructure using the Class primitive in the CIF. See IShell on page 50 for more information on IShell-based classes, which are in-process classes with certain characteristics.

The following is an example of declaring an in-process class in CIF:

```
Class {
   classid = AEECLSID_MyClass,
   newfunc = MyClass_New,
}
```

MyClass_New is the constructor of the in-process class written in C/C++ code and is invoked when the class is instantiated by IShell_CreateInstance() or IEnv_CreateInstance() on AEECLSID_MyClass. The class is instantiated in the same process (or more accurately, the Env) of the caller. newfunc is explicitly specified in CIF for MOD1 files. For MOD files, the constructor is set up by the helper files, e.g. AEEModGen.c

For more information on CIF, see the *Resource File and Markup Reference*.



Service class

Service classes are non-applet classes that service the caller's request in a designated process. They are also known as services and are similar to a Windows service or Unix daemon running in the background. Service classes were introduced in BREW 4.x and Brew MP, and are only supported in MOD1. Most Brew MP APIs are implemented as service classes, whereas most BREW APIs are in-process classes. See MOD1 on page 61 for more information.

A service class is essentially a code extension (to an applet) that is instantiated and executed outside the application context and outside the BREW Shell (or thread). The execution context for service classes can either be the kernel process or a server process, which is statically specified in the CIF of the containing module. A server process provides a separate execution context other than the BREW application (or applet) context. For more information, see System process model on page 18.

In Brew MP, any communication across the boundary of an execution context has to be performed through a remote invocation mechanism (invoked via a stub and skeleton code). Since a service class is always instantiated outside the application context, all calls from an applet to a service object are remote invocations. See Remote Invocations on page 16 for more information.

Service classes that provide QCM interfaces can be registered with the component infrastructure using the Service primitive in the CIF. For a CIF example, see "CIF example for a service class" below. For more information on CIF file format, see the *Resource File and Markup Reference*.

The service object is created in the designated process (kernel or server process). It can only be created in the Env of the process outside the BREW Shell and therefore any service implementation or classes it uses cannot use the IShell interface. In Brew MP, a service class cannot use static APIs such as IShell.

The privileges for a service object come from the hosting process. Methods invoked by a caller from the same Env result in direct function calls. Methods invoked by a caller from a different Env result in remote invocations. If the service object is created as a singleton, there is only one instance of the class in the entire system.

Uses of service classes

Service classes provide the following functionality:

- **Enable privilege separation and better security**

  While an in-process class is instantiated in the same execution context as its caller and therefore acquires privileges from the caller, a service class acquires privileges from its hosting environment (the kernel process or a server process). Each service class can also specify the privileges the caller must possess to access the service (see Security on page 22 for more information). Because a service object executes in a different execution context with its own set of privileges, Brew MP can provide privilege separation and more granular control of privileged operations.

  Privilege separation is a technique in which a program is divided into parts that are limited to the specific privileges needed to perform a specific task. For example, if full access to the file system requires privilege A, and any file can be deleted with that privilege, it is considered dangerous to grant privilege A to any application. Instead, file access should be managed and controlled by a trusted service class hosted in a process with privilege A. This service class then specifies that callers must have privilege B and exposes reduced file access functionality to them. Only privilege B needs to be granted to applications that need to gain file access (through the service class) instead of privilege A. More granular privileges or access policies can also be enforced with the use of IPrivSet in the service class. See the "Privileges" section in the *OS Services Technology Guide for Manufactures* for more information.
- **Promote higher fault tolerance**

  In legacy BREW, there is a single process environment in which there is only one BREW Shell (or thread) in the process that hosts all applets and extension objects used by the applets. There is a single execution and protection domain for all BREW objects and applets have unrestricted access to memory and resources. One misbehaving applet can potentially crash the device or make it unusable. An applet could potentially branch to any address in physical memory and read any area of the device memory or use any of the peripherals.

  When a service class is instantiated in a separate server process, the Brew MP operating system and any applets that call the service are protected from any potential faults or crashes that may be caused by the service object. Any misbehaving code inside a service class at worst causes its hosting process to terminate, not the operating system nor the applets that call the service.
- **Can make use of pre-emptive multithreading**

  Service classes can use pre-emptive multithreading because they are instantiated outside the BREW Shell (a single-threaded application environment). For an application to make use of pre-emptive multithreading, the portion of the functionality that needs to be preemptively multithreaded should be separated from the applet class and implemented in a service class. See *Using IThread1* on the Brew MP Developer Network for more information.

- **Enable data and resource sharing between applications**

  Each application runs in its own context (protection domain) and data or resources allocated by one application cannot be directly accessed by another application. To share data between applications, a singleton service class can be used. If a service class is instantiated as a singleton, there is only one instance of the service class running in the Brew MP system. This singleton service is single point of contact for managing and controlling access to data and can provide interfaces to allow data to be shared between applications.

CIF example for a service class

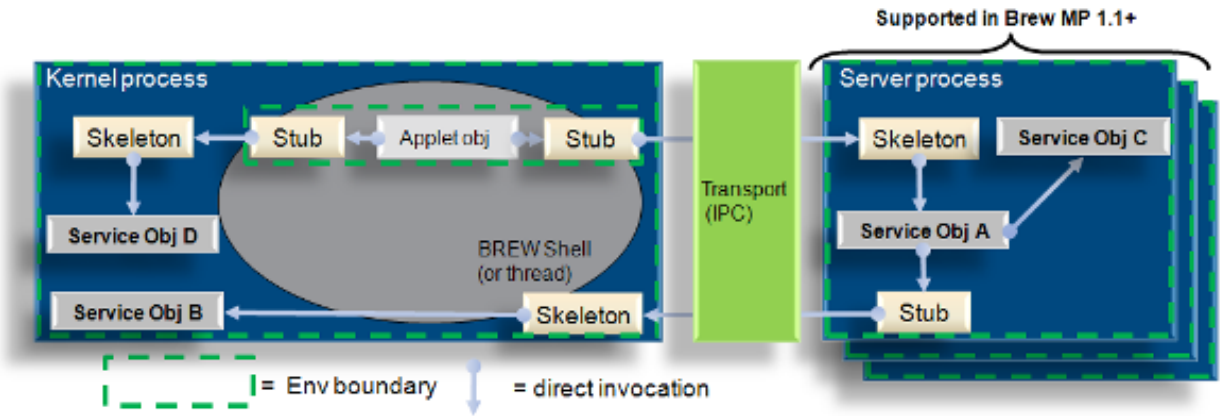
A service class is declared via the Service primitive in the CIF, as shown in the following example. Note that AEECLSID_MyClass is defined as a servedclassid for MyService, which is not done for the in-process class definition.

```
Service {
  serviceid = AEECLSID_MyService,
  iid       = AEEIID_MyService,
  serverid  = 0,
  required_privs = {0},
  servedclassid = AEECLSID_MyClass
}

Class {
   classid = AEECLSID_MyClass,
   newfunc = MyClass_New,
}
```

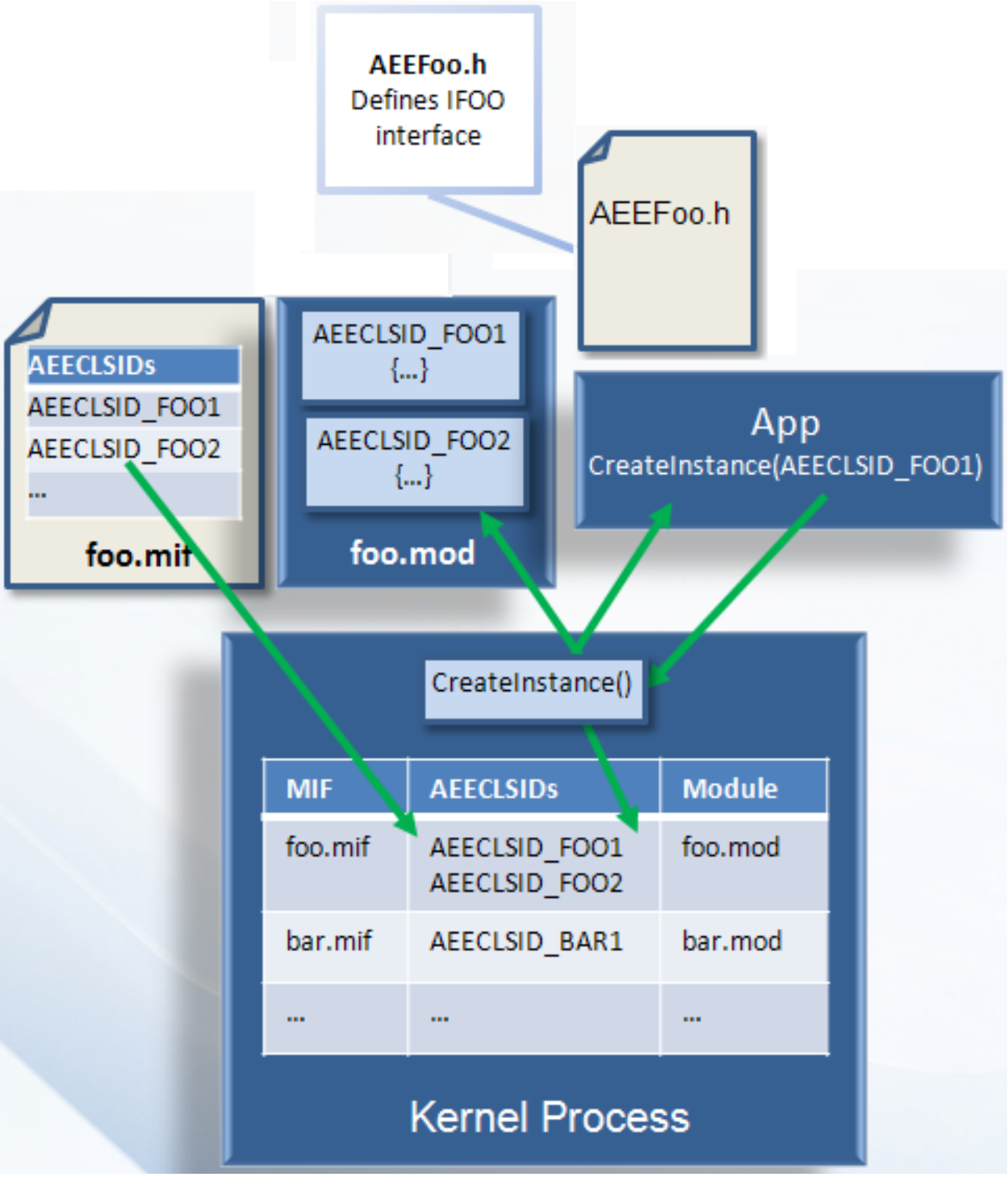
MyClass_New is the constructor of the service class written in C/C++ code and is invoked when the class is instantiated by IShell_CreateInstance() or IEnv_CreateInstance() on AEECLSID_MyService. serverid = 0 indicates that the service class is instantiated in the kernel process, and AEEIID_MyService specifies the default interface (defined in IDL and remotable) that the service class implements. For more information on CIF, see the *Resource File and Markup Reference*.

Service classes publish QCM interfaces that meet remotable criteria, also referred as Directly Remotable Interfaces (DRI). DRIs are interfaces for which a remote invocation framework can marshal data and objects on the invocations across the domain boundaries. Examples of domains are processes, processors, virtual machines, etc. Remotability must be maintained for any service implementation, since the caller cannot reside in the same protection domain (Env) as the service object. The remote invocations involve data marshaling and unmarshaling by the stub and skeleton code. If the invocations need to go across process boundaries, a transport layer to bridge the stub and skeleton code is also involved. For more information, see Remote Invocations on page 16.



Class resolution in Brew MP

The following diagram illustrates an example of class resolution in Brew MP.

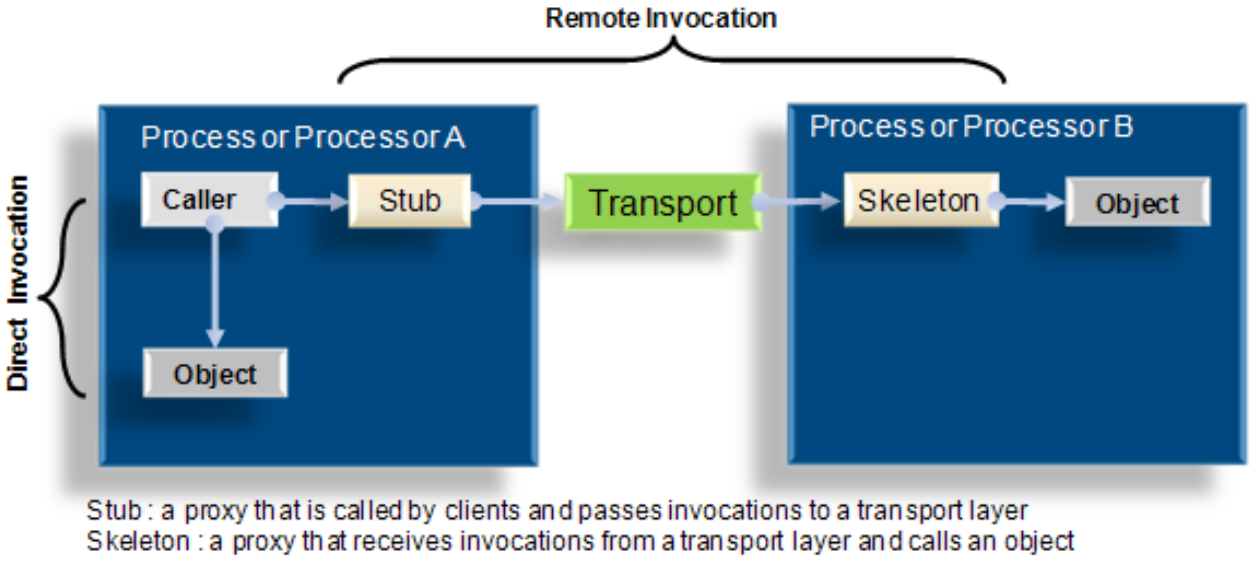


Class	Caller of the class
• The module's MIF (foo.mif) lists the supported ClassIDs and class system privileges. The system populates the ClassID to the module (foo.mod) table upon startup. • foo.mod contains the implementation (the classes) of the IFoo interface defined in AEEFoo.h.	• Includes AEEFoo.h, which defines the interface and can be used to invoke the functions exposed by the interface. • Requests the class that implements the IFoo interface by invoking CreateInstance with the ClassID of the class. The system resolves the class, locates the module that contains the class (foo.mod), loads the module, and instantiates the class, and the pointer to the instance of the class is returned to the caller.

Remote Invocations

A remote invocation is a mechanism that allows a client to invoke an object in a different execution and protection domain, such as a process, virtual machine, or CPU.

When a client invokes an object within its protection domain, it results in a direct invocation or direct function call. However, when the client needs to invoke an object from a different protection domain, it results in a remote invocation, as shown in the following illustration:



For a client to be able to trigger remote invocations and remotely invoke an object in a different protection domain, the object needs to support interfaces that are remotable. The remotability of an interface relies on the availability of the proxy code associated with the interface to marshal and unmarshall the data and requests going across protection boundaries. Such proxy code in Brew MP consists of stub and skeleton code. If the protection domains that separate the client and the object are processes or processors, there is also a transport layer that bridges the communications between the stub and the skeleton.

Remote invocations allow the object and its caller to be hosted in different processes, which is the basis of the multi-process execution environment Brew MP is designed to support. The caller does not need to know where the actual object is created and invoked. For the caller, invoking an object in a different process is no different from invoking an object within the same process. Any invocations across protection

boundaries need to be granted; the callers need to have sufficient rights or privileges to access the objects.

Components and modules

Components

Brew MP is a component-based framework. Applications and services used by the applications are essentially various types of components. A component is a logical concept of one or more classes that are self-contained and allow for dynamic linkability and inter-changeability at the binary level.

If the implementation of a class has to be statically linked to other pieces of software outside of the class to perform certain functionality, that class itself does not qualify as a component. However, the class along with other pieces of software together can be built into a component if those pieces of software don't have any other external static dependencies. In other words, software within a component can be tightly coupled and statically dependent on one another.

Modules

Brew MP modules are the fundamental unit of code loading. A module is an executable binary file that consists of one or more components compiled into a single image. The module is a single point of entry for the AEE shell to request classes owned by the module.

They can be statically linked (.lib), or stored in the file system (EFS) as dynamically loaded modules (see .mod, .mod1 on page 61).

- All dynamic modules are digitally signed
- Dynamic modules are loaded into RAM when needed and unloaded when no longer in use

APIs are exposed by modules as objects associated with interfaces and classes. Each module can contain implementations for one or more classes, and must have a corresponding MIF associated with it. The MIF contains information about the contents of the module, such as supported classes, supported applets, applet privileges, and applet details (like the title and icon). The MIF also contains unique ClassIDs for each of the module's classes, and specifies which classes are exported for use by other modules. See the *Resource File and Markup Reference*, and the *Tools Reference* on the Brew MP Dev Net for details on creating new MIFs.

Runtime environment

The following sections discuss Envs, the system process model, registry support, and inter-application communication.

In a deployed Brew MP device, the runtime configuration is expected to be a number of user processes hosting various applications and services (see user processes on page 18), and a single kernel process hosting the component infrastructure and privileged services. In this Brew MP environment, applications are implemented as applet objects, which are loaded and executed in the BREW application framework (BREW Shell). To perform various tasks, applications can leverage utilities or services available on the platform to access additional functionality. Most of the generic utilities or services are already exposed to Brew MP applications through platform APIs (see the *C/C++ API Reference*). Any additional services can be implemented and provided in the following ways:

1. Provide the utility as an in-process object. For example, providers of an image decoder such as a software JPEG decoder can provide an implementation of the IImageDecoder interface that is directly instantiable in the context of the Brew MP application that needs this functionality.

2. Provide the utility as a Brew MP service, an advertised remote object. This service could be hosted in a server process that has access to a hardware JPEG codec or DSP and is able to provide JPEG decoding service to any Brew MP application or any other client running in any process that has the privileges to use this service. A server is a process that hosts advertised objects. A service resource (see the *Resource File and Markup Reference* included with the Brew MP SDK) identifies which server hosts the object, which class should be used to instantiate the object in that host process, and any privileges that are required to access the service.

Environments

In Brew MP, an environment (or Env) establishes an execution domain (or context) for each object in the system.

Every object in the system reside in an Env. Objects residing in the same Env are considered local to each other and can be invoked and accessed directly from each other. They share the same privileges and access the same singleton instance created in the same Env.

Env manifests itself as an object that supports IEnv. When a class receives an IEnv pointer, it is the object that the IEnv pointer points to that determines the execution domain for the object of the class. For more information, see IEnv on page 50.

There is one Env per applet and per process. Brew MP supports multiple applets running in the same process, so each applet applet has its own Env, with its own privileges, which is separate from the Env of other applets in the same process. The Env determines the execution domain for an applet. The memory protection domain is established by the Env of the process in which the applet resides. Applets in the same process still share the heap memory in the memory protection domain.

System process model

Brew MP employs its system process model to host services and applications. A Brew MP process defines the set of rights and restrictions for the execution of the code it governs to access memory or other resources.

In BREW, applets execute in a single designated thread on the handset. This thread is commonly referred as the BREW thread, and is a single task context shared by all the running applets. Most of the earlier software implementations for BREW were created with the assumption that all the users of the software would execute in the same thread.

Threads live in processes, and code executing in a thread is limited in its access to memory and kernel services according to the process in which it lives. All operating system services of Brew MP are represented by objects. For example, critical section functionality is provided via an object that implements the ICritSect interface. The kernel enforces that user processes can only access objects that they have been granted. The kernel does not decide to whom the objects are granted, nor does it dictate what the objects can do. The kernel, therefore enforces the mechanism, and not the policy.

Each process is given an Env that maintains its rights and establishes the protection boundary for the process. The protection domain and rights of a process are maintained via its Env. See Env on page 18 for more information.

The Brew MP system process model transparently supports memory-protected (multi-process), non memory-protected (single-process), single processor and dual processor implementations. There are two kinds of processes in which Brew MP objects can reside, the kernel process and the user process. In Brew MP, a running system consists of threads executing in the kernel and some number of user processes.

For more information, see kernel process on page 19 and user process on page 19.

Kernel process

The Brew MP kernel process is a privileged execution environment that can access all memory, controls user processes, enforces privileges, and controls process interaction. There is only one instance of the kernel process on a Brew MP device.

The kernel process is the utmost privileged process that governs all the user processes and enforces their rights to access memory and resources. It stores data structures that keep track of the object references held by each process (see data structures on page 30) . An object table is maintained for each remote object used by a process. By governing which remote objects the process can invoke or pass to other processes, the object table enables the system to perform cleanup when the process is killed. The kernel process also hosts critical system services.

User process

A Brew MP user process is a confined execution environment that can only access memory and system (kernel) resources (files, devices, etc.), for which the process has been granted access. User processes do not have any inherent authority over the system. They only have objects they have been granted, and all kernel requests operate on objects.

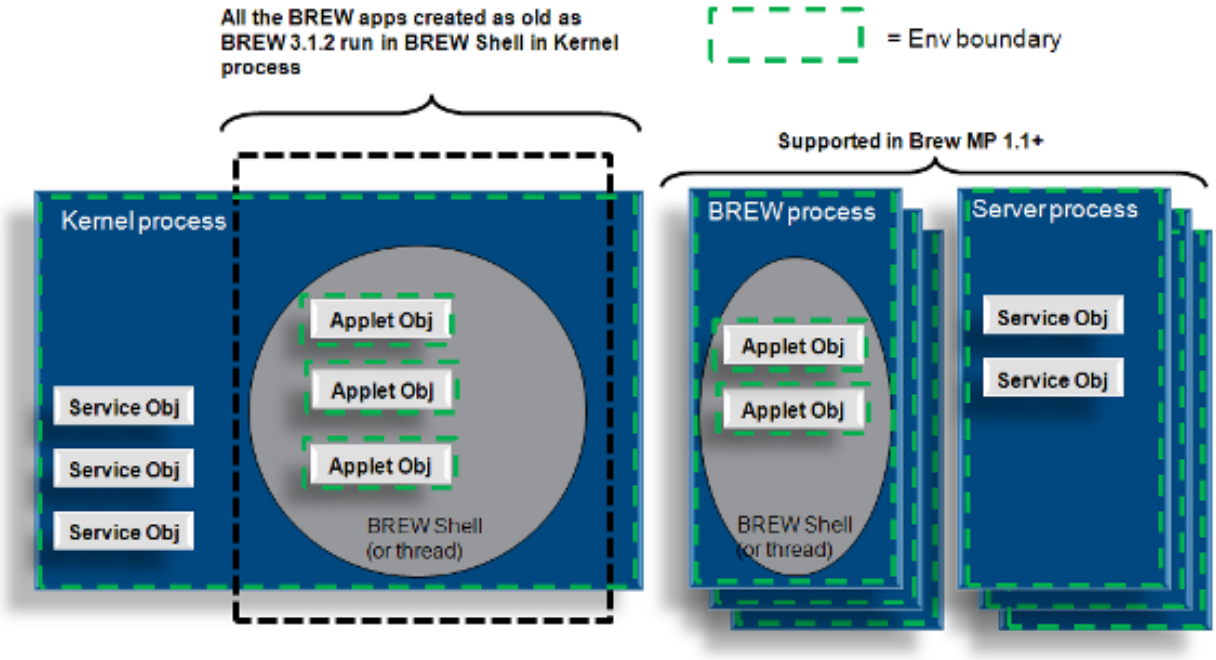
There are two types of user processes, depending on the type of classes they host:

- BREW process - (supported in future versions of Brew MP) maintains a BREW Shell that hosts only BREW applets as well as any objects the applets create in-process
- Server process - only hosts service objects

Brew MP supports pre-emptive threads whose memory and service access is governed by their host process.

In Brew MP, services are transparently invoked between processes by way of remote invocations (through generic stubs and skeletons). This mechanism allows caller and service implementation code to remain constant across single process, multi-process, and multi-processor environments.

The following shows the BREW runtime environment, including kernel processes, BREW processes, and Server processes.



Registry support

Brew MP provides a system-wide registry, which associates interfaces and classes with string-based registry keys.

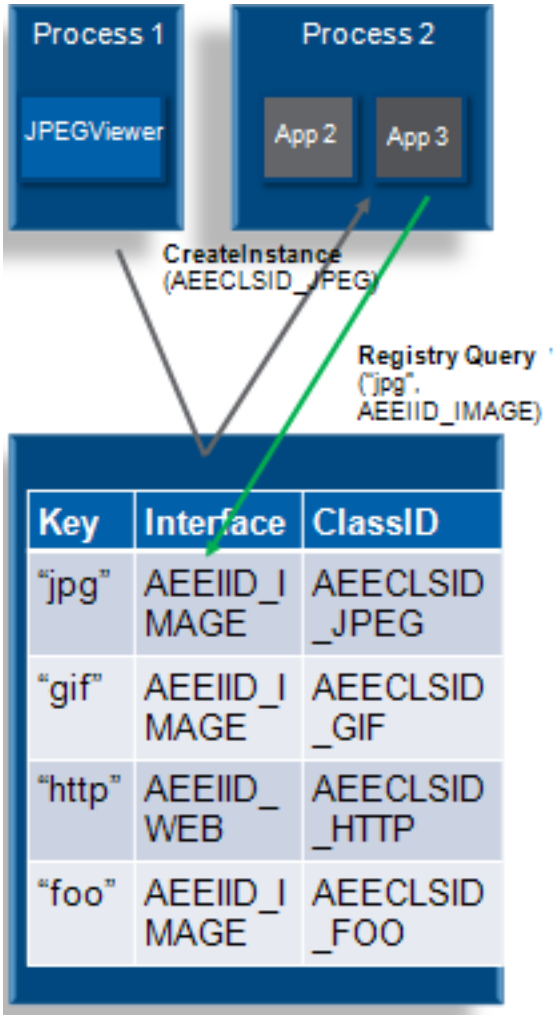
Common uses include multimedia and network protocol handlers (.jpg, .png, .http, etc.).

Searches are qualified by an associated interface (AEEIID), which enforces returning classes (AEECLSIDs) that map to the expected definition. The queries can be made via ISHELL_GetHandler(). See the *C/C++ API Reference* for more details.

The system registry is generally populated at startup with registry entries specified in the digitally signed MIF of their handlers. These entries are loaded in the order that the MIFs are loaded and parsed.

Registry values can also be updated at runtime. There are few restrictions on which entries can be updated at runtime. However, these updates are not persistent and need to be repeated, if necessary, each time the system is started.

Queries return an AEECLSID, which is then used to obtain the associated implementation.



MIME Type

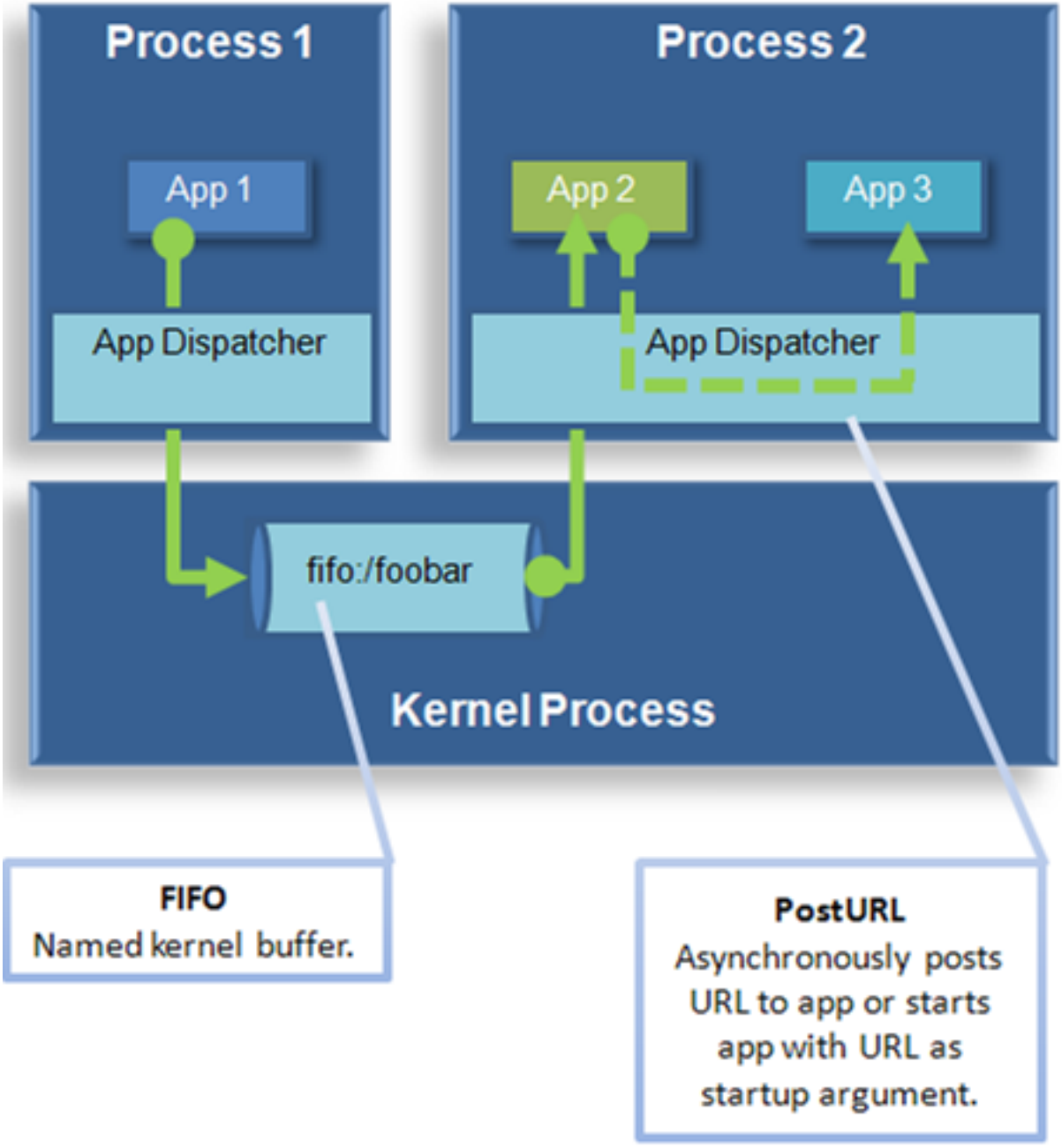
All objects in Brew MP are stored in a binary resource file with the associated MIME Type string. An object resource always has a MIME type associated with it. MIME types are similar to file extensions, used to identify the type of data a file contains. MIME Type registry entries are specified through the SysRsc primitive in the CIF. For more information, see the *Resource File and Markup Reference*.

Inter-application communication

Brew MP provides several mechanisms for inter-application communication. One is a system-level FIFO mechanism that allows processes to write to and read from named kernel memory buffers. Access control for the FIFO buffer is specified in the CIF through the FIFO_ACL_Grant primitive. The data can take any format and the API follows a familiar asynchronous I/O model. See the *Resource File and Markup Reference* for more details.

Brew MP also supports local loopback sockets. Similar to FIFO buffers, this mechanism employs the familiar socket I/O paradigm with the data transmitted between applications and processes rather than over the network. As with FIFO buffers, the format of the data is private.

Applications can asynchronously dispatch URLs to other applications by way of ISHELL_PostURL and ISHELL_BrowseURL. PostURL queues an event which later causes the application to be loaded. The application can then choose to start if desired. BrowseURL synchronously loads and starts the associated application with the URL. Brew MP supports a number of pre-defined URLs.



Security

Security covers running applications in user mode, privileges, and Access Control Lists (ACLs).

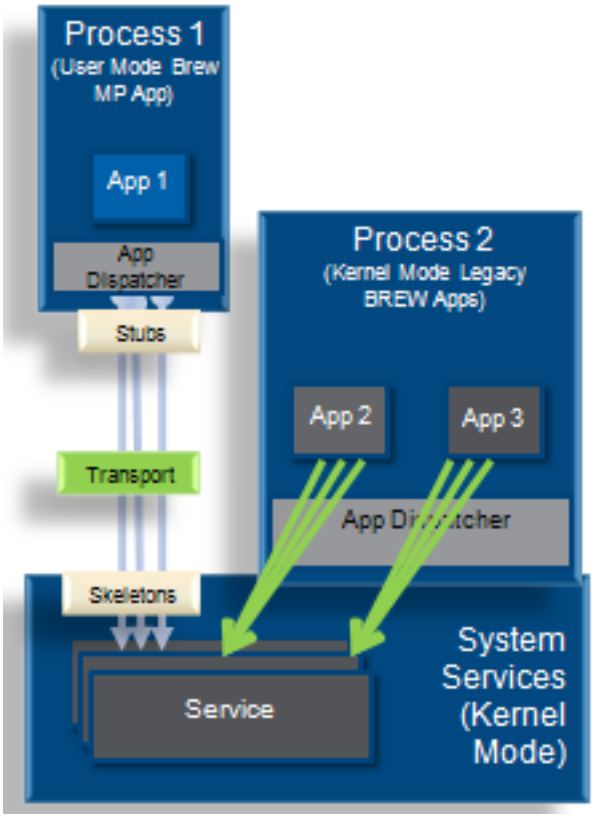
User mode and kernel applications

The App Dispatcher framework allows Brew MP to host applications in user mode.

A user mode application runs in its own protected memory space. Programs running in user mode can only interact with other programs that are user mode accessible. All service classes are user mode accessible, though subject to privilege validation. Running applications in user mode requires that all APIs used by the application are supported in user mode in one of two manners:

- In-process classes that are shared components (for example, Crypto)
- Service classes that can be remotely invoked (for example, GPS)

It is important to note that in Brew MP 1.0, not all service APIs are supported across process boundaries. A user mode application does not have access to the entire API set in Brew MP. Some APIs are not safe to use in user mode and are disabled. Existing Brew MP applications requiring services not currently supported in user mode are loaded into a shared kernel mode process. An application runs in kernel mode by default, unless the application's MIF specifies running in user mode.



Not all in-process classes are user-mode accessible. For an in-process class to be user mode accessible, it should meet the following criteria:

- User-mode dependency ready: Any classes it depends on should either be service classes or other user-mode accessible in-process classes.
- User mode reference ready: All the memory regions the object uses are negotiated by the OS for use in that process runtime. This means no read and write references to non-const data of static

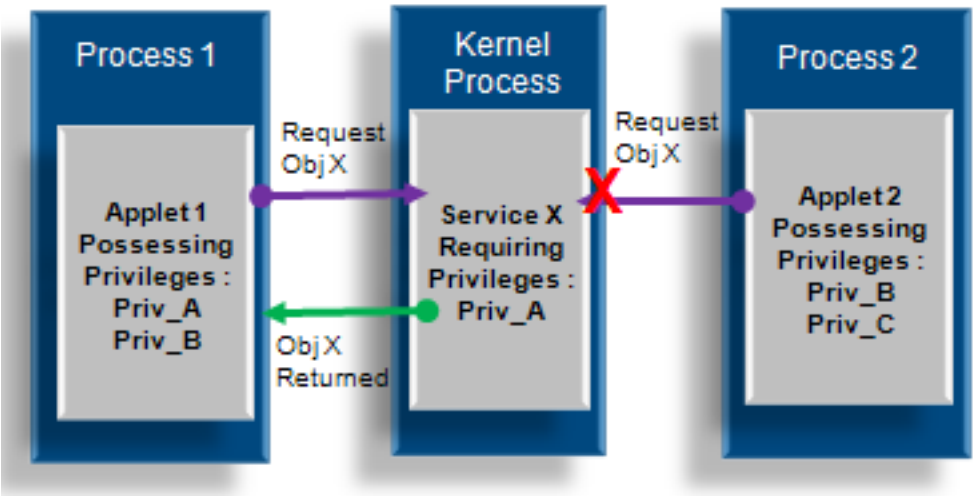
storage classes, and no branches or static links to code that makes these references (which would indirectly make references to the non-const static variables).

Privileges and ACLs

Brew MP's security model is based on least privileged execution. Processes and applications can access only the services for which they have been granted access.

Privileges are used to control access to APIs. Brew MP provides Access Control Lists (ACLs), permitting modules to share access to their private directories. ACLs are used in an application to allow a given file or directory to be accessible to other applications.

Privileges are dynamic. They are not hard coded into the system. New privileges can be defined and associated with any new interface. This allows OEMs or third party developers to protect system-critical functionality without modifying the core Brew MP implementation.



For more information on privileges, see privileges on page 31.

Application UI model

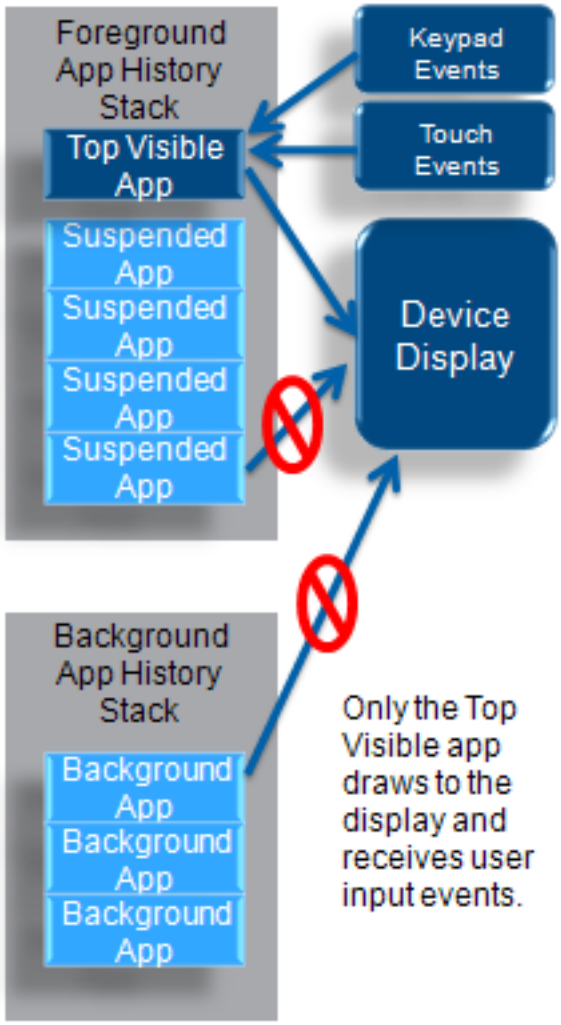
Brew MP implements a top visible application UI model, where only one application can draw directly to the display.

When an application is successfully started (EVT_APP_START), it becomes top visible or in the foreground. It then receives all keypad and touch events and can draw directly to the display.

If another application is started, the system attempts to suspend (EVT_APP_SUSPEND) the previously top-visible application. If the application handles this event, it remains loaded and can still perform all normal tasks. It won't receive keypad or touch events and cannot draw to the display. Applications that do not handle EVT_APP_SUSPEND are unloaded.

Brew MP maintains a list of applications with the top-visible application at the top of the list and the suspended applications in order below. When the top-visible application is closed (EVT_APP_STOP), Brew MP resumes (EVT_APP_RESUME) the next previously suspended application. If that application was unloaded, it is restarted.

Applications can also place themselves in the background. These applications are not in the application history stack, and only come to the foreground if they are started.



A running application can be in one of three states:

1. Top-visible/foreground: There is only one top-visible application in Brew MP at any given time.
2. Suspended: There can be many. A suspended application can be resumed when the application that caused it to be suspended has terminated.
3. Background: There can be many, and they run in the background.

For more information on suspending and resuming applications, see the *Application Management and Technology Guide for Developers*, on the Brew MP Dev Net, as well as event handling on page 33 in this document.

UI Widgets

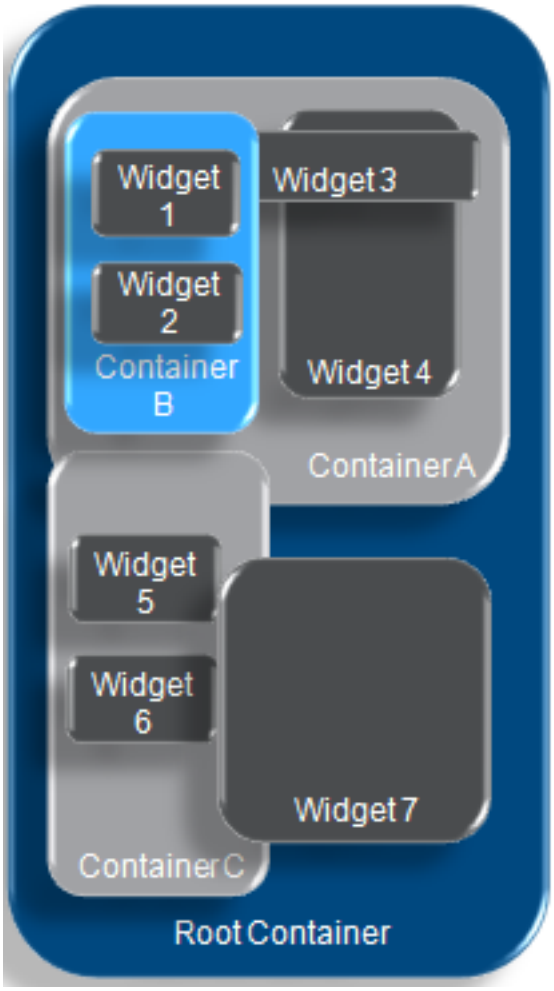
Brew MP UI Widgets is a C-based framework for creating UI applications.

Each widget represents a visible element on the display, and lives inside of a container that manages the layout, z-ordering, focus, and event-routing for the widgets and other containers that it contains.

Widgets draw to the display through a cooperative invalidate-draw cycle. When a widget needs to update its view, it invalidates its content. The invalidation makes its way up through the container hierarchy until reaching the root container, at which point the draw cycle is initiated.

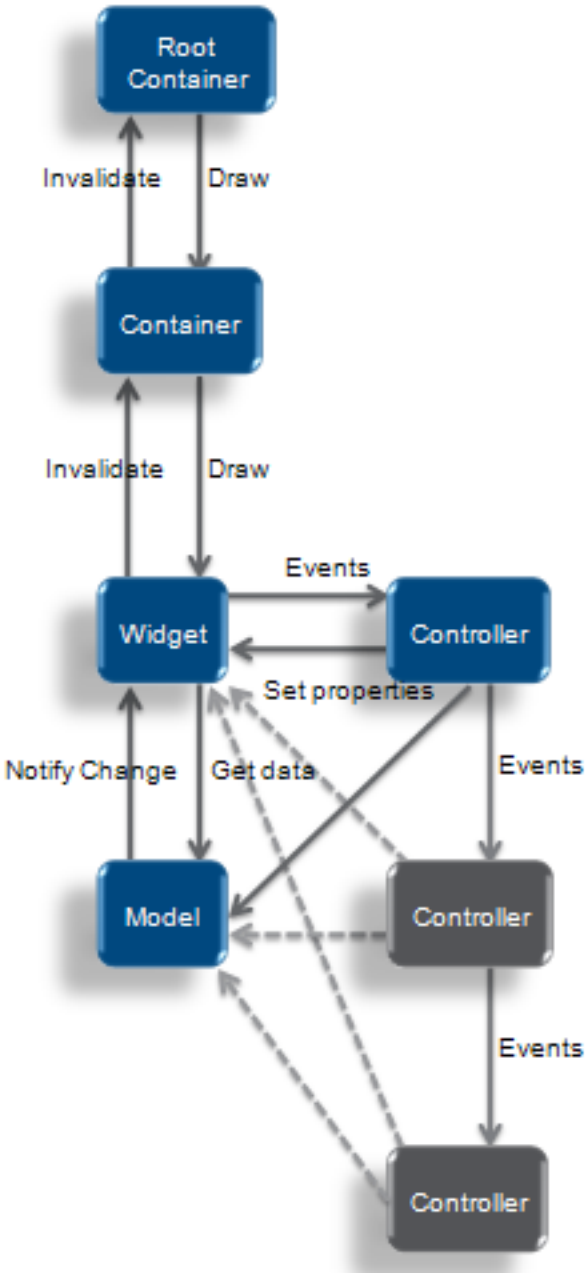
The draw cycle controls which widgets draw their content, and in what order. It preserves z-ordering of all elements, and only redraws those widgets that need to be redrawn.

Widgets represent the basic UI drawing model in Brew MP that is leveraged by the Brew MP window manager as well as the Flash and Trig application models.



Brew UI Widgets are loosely based on a model-view-controller (MVC) pattern, which separates UI, controller logic, and data. In many cases, widgets combine the view and controller in a single object, but keep a separate model.

While widgets have default built-in and/or associated controllers to handle events, it is also possible to attach additional controllers (event handlers) to a widget to perform custom logic.



Some widgets are used with multiple different models to achieve different goals. For example, Frame Widget can accept media frames from a camera viewfinder (CameraFrameModel), a video player (MediaFrameModel), or other source that provides frame-based data for playback.

Widgets support properties that are exposed to controllers and applications through get/set methods. Each widget supports properties that are specific to its function, in addition to common properties such as location and extent. The visual appearance of a widget can be vastly modified by changing its properties.

For more information, see the *Widgets Technology Guide* on the Brew MP Dev Net.

Windowed application model

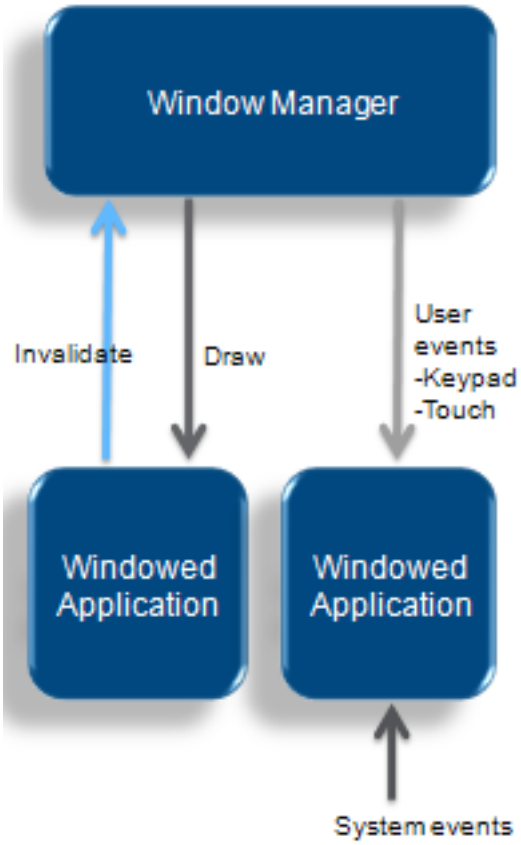
The Brew MP Window Manager application allows multiple applications to share the display.

When using windowed applications, the Window Manager is the top-visible application and the windowed applications run in the background.

Because a windowed application is running in the background, it does not receive user events, such as key and touch events, to its IApplet interface. Instead, the window manager forwards user events to the window that currently has focus (for keypad events), or to the top-most window that was touched (for touch events).

Non-user events (URLs, notifiers , etc.) are still sent to the windowed application's IApplet interface.

Windowed applications draw cooperatively through the window manager application using Brew MP's widget draw cycle. A window that needs to update invalidates itself rather than drawing directly to the display. The window's draw function is called by the window manager at the appropriate time to interleave the drawing with other windows in the system.



For more information, see the *Window Manager Technology Guide for Developers*, on the Brew MP Dev Net.

C/C++ application structure

The Brew MP SDK includes plugins that integrate with the Visual Studio and Eclipse development environments to simplify many of the tasks performed when creating Brew MP applications. The plugins

include a wizard to generate an application framework for a Brew MP C/C++ application or extension that provides a baseline project complete with comments. Required source files are included in the project. The wizard also adds the header files into the #includes section. Additional header files are added as the application is built upon. The default application framework created by the wizards is for MOD1/DLL1.

MOD1 project setup

The setup for MOD1 applications varies from that of MOD applications. The entry point for MOD1 applications is IMod_New(), rather than AEEMod_Load(). CIFC generates the module IMod stub, as well as code to call the individual classes' New() functions, eliminating the need to write your own module entry point.

For code in a MOD1 file:

- The IEnv object is the first object provided to each class upon creation.
- Rather than using AEEModGen.c, the CIF compiler (cifc.exe) generates code to provide the module entry point.
- Only classes running inside the BREW Shell have access to the IShell object, and use it to discover and create other objects available in the system.
- Heap memory is allocated via IEnv_ErrMalloc().

  Avoid using AEEStdLib.h functions, including MALLOC, GETUPTIMEMS, and STRNCPY. Some AEEStdLib.h functions can be replaced with functions from AEEStd.h. Malloc can be replaced with IEnv_ErrMalloc(). Uptime functionality can be replaced with ISysClock functions. Timer functionality is provided by ISysTimer.

The following is an example of the MOD1 entry point, which takes an IEnv* parameter rather than an IShell* parameter. This piEnv is passed to the New() function for the class via IMod_CreateInstance(). The class structure can be changed to hold this Env pointer instead.

```
int IMod_New(IEnv *piEnv, AEEIID iid, void** ppiModOut)
```

MOD project setup

Along with the application source .c file, two other helper files are included:

- AEEAppGen.c - Defines a Brew MP application and provides general application functionality such as event handling.
- AEEModGen.c - Defines a Brew MP module, loads it in memory, and provides access to the module.

Without these helper files, you would have to define your module and applet on your own. Look through these files for insight into how the module and application are created.

For code in a MOD file:

- The IShell object is the first object provided to each class upon creation.
- The IShell object is always available, and can be used to create and discover other objects available in the system.
- Both MALLOC() (from AEEStdLib) and IEnv_ErrMalloc() can be used to allocate heap memory.
- MOD applications inherit IApplet, which provides a mechanism to the shell to pass events to an applet.

The following is an example of a MOD entry point.

```
int AEEMod_Load(IShell *piShell, void *ph, IModule **ppMod)
```

HandleEvent needs to be implemented when the class supports IApplet, which can be the case for both MOD and MOD1.

For more information on MOD vs. MOD1, see the *MOD vs. MOD1 in Brew MP* Technology Guide on the Brew MP Dev Net.

Coding

As a suggested approach to application development, consider an application state machine framework to simplify application development and improve the efficiency of user interfaces.

In this state machine framework, each component, screen, or function of your application can be built as a state. Each state can then be stacked with other states, allowing the application to move freely through the stack. For instance, a menu is activated thus activating the menu state. This menu state is pushed onto the stack by the application. When a menu option is selected and the command is processed, this in turn activates a new state on the stack. If the clear key is pressed while the menu is activated, the menu state could be popped off the stack, thus presenting the previous state.

Data structures

The data structures used by Brew MP API functions define the format and content of the input/output data passed between the functions and applications.

API data structure types

Most data structures are specific to a particular Brew MP interface, and their type definitions are contained in the header file for that interface. More general, widely used data structures are found in the AEE.h file. The description of each function contains links to the descriptions of all relevant data structures.

The following are API data structure types.

- Structures and unions - Many Brew MP functions take pointers to structures as input parameters. To use these functions, an instance of a structure is populated, and a pointer is passed to the instance when calling the function. For example, the IGraphics shape-drawing functions have structures as input parameters that define the dimensions of the shape to be drawn. Many Brew MP functions return pointers to structures as output. The IFile and IImage interface functions, for example, return information about files and images in structures.
- Enumerated types - Many Brew MP variables and structure members take on values from a finite set defined by the C typedef enum construct. For example, the font types supported by the IDisplay interface's text-drawing functions are specified with an enumerated-type definition.
- Constants - The Brew MP API functions make use of a number of constants defined with the #define directive. For example, all of the Brew MP event codes are defined in this manner in AEEEvent.h.

API helper functions

The various helper functions provided by the AEE include string functions, functions in the standard C library, utility functions, and other items. Standard C library refers to the ANSI standard C library supplied with C/C++ compilers/Integrated Development Environments.

Some of the helper functions offered by AEE are wrappers that directly call the standard C library functions to do the following:

- Eliminate unnecessary linkage with the standard C library. When there are multiple applications loaded on the device, each application carries the extra baggage of the standard C runtime library.

To avoid this, AEE maintains a single copy of the standard C library. All applications can make use of this copy. Applications must not make direct calls to the standard C library functions, so that the static C runtime library does not become part of the binary image.

- Eliminate static data in dynamic applications. Linkage to standard C library functions can introduce static data into an application, preventing it from being dynamically loadable.

Applications must not directly invoke the standard C library functions (memcpy(), for example). Instead, applications must use the helper functions provided by the BREW AEE such as MEMCPY(). A distinct difference between the helper functions and the rest of the AEE functions is that interface-specific information is not needed to access the helper functions.

BREW applet structure

The applet structure is the data definition of a Brew MP application. Interface pointers, large buffers, and global data should be included here. The following is the applet structure provided by the Brew MP Plugin Wizard.

```
typedef struct _myapp {
   AEEApplet  applet;    // First element of this structure must be AEEApplet.
   IDisplay * piDisplay; // Copy of IDisplay Interface pointer for easy access.
   IShell   * piShell;   // Copy of IShell Interface pointer for easy access.
   AEEDeviceInfo  deviceInfo; // Copy of device info for easy access.
   // Add your own variables here...

} myapp;
```

The following elements are included in the Wizard-generated applet structure.

- AEEApplet must be the first element in the structure. AEEApplet is defined in AEE.h as a BREW applet type. Placing this element in the first position defines the address of the applet with the same address as the data structure for the applet. In the code above, "applet" is declared as an AEEApplet type.
- IDisplay is a primary interface used by all applets for rendering text and other information to the display. The pointer *piDisplay is declared here to make the instance of the IDisplay interface available to all applications.
- IShell is an interface used by all applications. The pointer *piShell is defined here for access to shell services throughout the application.
- AEEDeviceInfo is a BREW data structure defined in AEEShell.h, used for holding device configuration and capability information. The variable "deviceinfo" is declared as this type.

Additional interfaces are added here, along with large buffers, global variables, and other data that needs to be allocated at application creation.

Privileges

Privileges imply rights or restrictions to access resource and/or objects such as memory access, CPU time, and platform services.

Privileges are represented by 32-bit unique IDs (e.g. AEEPRIVID_XXX) and listed in the MIF. The kernel maintains the privileges list for each object in the system to ensure processes or applications can only access the objects to which they are given access. Access to services is controlled by privilege sets maintained in the kernel. A privilege set is an extensible array of 32-bit AEEPRIVID values. The AEEPRIVIDs correspond to privileges the requestor can obtain to gain access to certain resources. The privilege set for a process or application is contained in its digitally signed MIF (module information file). The kernel process establishes the privilege set for the process or application when it is loaded.

In the following example, both Applet 1 and Applet 2 intend to instantiate and access Service X. Since only Applet 1 has sufficient privilege, Applet 1's request is granted and Applet 2's is denied.

To grant address book privileges, write the following code in the CIF, and compile the CIF to generate the MIF.

Be sure to include the following to grant privileges:

```
"AEEPLPrivs.bid"
```

The AEEPRIVID_PLFile privilege in the CIF/MIF must be defined before the Call History database can be accessed.

```
Applet {
appletid       =  AEECLSID_CALLHISTORYAPP,
   resbaseid      =  1000, -- Applet base resource id
   applethostid   =  0,
   privs          =  { AEEPRIVID_PLFile },     // AEEPRIVID_PLFile privilege
   type           =  AEE_AFLAG_HIDDEN
}
```

Acquiring/possessing Privileges

For applet objects to possess privileges, specify the following in the CIF:

```
Applet{
   ...
   privs = {AEEPRIVID_XXX},
}
```

For service objects to possess privileges, the privileges come from the hosting process. If the service object is in the kernel process (Service.serverid = 0), it has the same privileges as the kernel. If it is in a particular server process (Server.serverid = AEECLSID_SERVERSOMETHING), the server declaration is similar to the following:

```
Server{
   …
   privs = {AEEPRIVID_ZZZ},
}
```

For in-process objects to possess privileges, the privileges come from the caller. Privileges aren't specified for in-process objects.

Requiring Privileges

A service class or in-process class may require certain privileges that the caller must possess before they can be instantiated.

- An in-process class can be declared to be a privileged class (as in the following) so that the caller must possess the in-process class's ClassID as its privilege before it can instantiate the object:

  ```
  Class{
     ...
     privileged = TRUE,
  }
  ```

  If the Class.privileged field does not exist in the declaration, no privileges are required to instantiate this in-process class.
- A service class can be declared to require certain privileges so that the caller must possess one of those privileges before it can instantiate it, as shown in the following example:

```
Service{
   ...
   required_privs = {AEEPRIVID_ABC, AEEPRIVID_EFG}
}
```

Service.required_privs = {0} designates that no privileges are required to instantiate this service.

- If the Service.required_privs field does not exist in the declaration, the service ID of the service becomes the default privilege the caller needs prior to instantiating the service.
- If the class specified by Service.servedclassid is a privileged class, the caller needs to possess its ClassID as its privilege prior to instantiating the service.

For more information on privileges, see the following:

- The *Settings Technology Guide for Developers* on the Brew MP website includes information on setting ACLs.
- The PrivLevel section of the *Resource File and Markup Reference*.
- AEEShell.h in the *C/C++ API Reference* (System > Services > Privileges)
- The section on managing applets in the *Resource Manager Help* (also contained in the *Tools Reference*)
- The privileges section of the *OS Services Technology Guide for Developers*.

Event handling

After an application is loaded, it receives all input via events. These events are received by the HandleEvent() function of the application.

When Brew MP passes an event to an applet, the event is handed off to the application's main event handler. This handler can handle the event and return, or pass the event to another handler, such as the root container's IWidget interface. In any case, the applet (and subsequently any widgets to which the event was passed) indicates whether it handled the event by returning TRUE (handled) or FALSE (not handled).

Note that as a simple event-driven environment, Brew MP demands that events are handled in a timely manner. An applet is expected to quickly handle the event and return. Some events are required system events that must be handled by the application. System events include application startup and shutdown as well as telephone and SMS interruptions. Failure to handle system events can cause the device to function improperly.

This section covers the following event handling topics:

Event handling concepts

IApplet is the event-handling interface that implements services provided by an applet. All applets must implement IApplet. IApplet is called by the shell in response to specific events. IApplet provides a mechanism to the shell to pass events to an applet. When the shell responds to an applet, it calls IApplet_HandleEvent(), and passes event-specific parameters to the function. This includes an event code (ecode), defined in AEEEvent.h. Based on the event code, word (wParam) or doubleword (dwParam) event-specific, context-sensitive data may also be sent.

IApplet_HandleEvent() is only called by AEE Shell. To send events to other applications, use IShell_SendEvent().

The following is a prototype of the main applet event handler:

```
boolean myapp_HandleEvent    //Main Event Handler
(
   myapp * pMe,               //pointer to applet
   AEEEvent eCode,            //event eCode
   uint16 wParam,             //context sensitive short param
   uint32 dwParam             //context sensitive long param
)
```

There are three event-related inputs that applets receive, passed in as the second, third, and fourth parameters of the HandleEvent() function.

- AEEEvent specifies the event code received by the applet. EVT_APP_START, EVT_KEY, and EVT_ALARM are examples of events received by applets.
- The third parameter is a short word value, which is context-sensitive and dependent on the event. There may or may not be a wParam related to the event.
- The fourth parameter is a long (doubleword), context-sensitive value that is dependent on the event. This can be a bit modifier, constant string, or other long value that is dependent on the event. There may or may not be a dwParam related to the event.

See the *C/C++ API Reference* for details on the wParam and dwParam parameters for each event.

Event types

The following are some event types used in Brew MP. These event categories classify the type of event based on where the event was generated.

Applet events are events generated by the shell for applet control:

- EVT_APP_START
- EVT_APP_STOP
- EVT_APP_SUSPEND
- EVT_APP_RESUME
- EVT_BROWSE_URL
- EVT_APP_START_BACKGROUND
- EVT_APP_MESSAGE

AEE Shell events are events generated by the shell:

- EVT_NOTIFY
- EVT_ALARM

Device events are generated by device state changes:

- EVT_FLIP
- EVT_HEADSET
- EVT_KEYGUARD
- EVT_SCR_ROTATE

User events are private to the application. Developers can define their own private events within the range starting at EVT_USER:

- EVT_USER

Touch events are generated by touch-enabled devices:

- EVT_POINTER_DOWN
- EVT_POINTER_UP
- EVT_POINTER_MOVE
- EVT_POINTER_STALE_MOVE

Special events include EVT_APP_NO_SLEEP, which is sent to an applet after long periods in which the applet is running timers but the user is not interacting with the device. Brew MP sends this event to the applet to check whether to allow the device to enter power-saving mode, usually at a slower clock rate. If the applet responds by returning TRUE, Brew MP does not allow the phone to enter low power mode. Note that returning TRUE results in shorter battery life; applets should use this capability sparingly.

For a comprehensive list of events, see the header file AEEEvent.h, and the AEEEvent structure in the *C/C++ API Reference*.

Critical events

When implementing an applet, handle only the events your applet might want to process. Some events can be ignored, such as in a game that uses only up, down, left and right keys as input, an event corresponding to a keypress of keys 0-9 can be ignored. Critical events received by an applet can't be ignored, regardless of the state of the applet. Pay careful attention to receiving all the critical events in any given state of the applet. Some events are not sent to the applet unless it specifically indicates that it wants such notifications. Applets must register for these notification events either permanently in the MIF, or dynamically using ISHELL_RegisterNotify().

The following events should be handled. The applet is closed if TRUE is not returned.

- EVT_APP_START
- EVT_APP_START_BACKGROUND
- EVT_APP_SUSPEND
- EVT_APP_RESUME
- EVT_APP_STOP

Event delegation flexibility

The event delegation model provides a great deal of flexibility. You may handle the event before and/or after delegating the event to an active widget. This provides the ability to do additional processing behind the scenes, and even to override or customize the behavior of a widget.

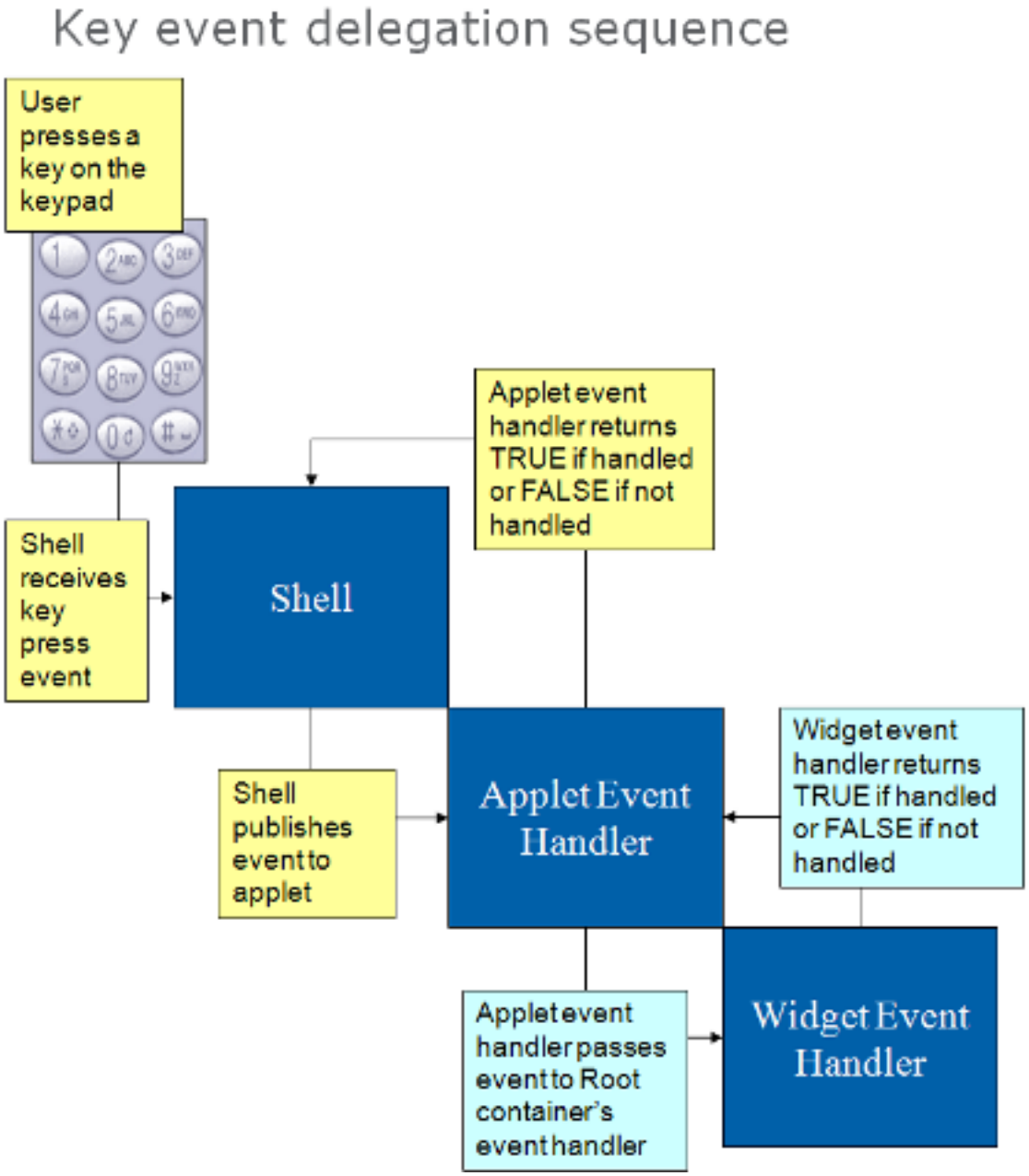
Key event delegation sequence

The sequence of a keypress event is illustrated below.

1. The process starts with the user pressing a key on the device.
2. The OEM software sends the key events to Brew MP.
3. Brew MP sends the event to the top-visible applet via the applet's IAPPLET_HandleEvent() method.
4. The applet can optionally handle that event first.

   If processed, the event handler returns TRUE. If the event is not handled the handler returns FALSE.
5. The event can be passed to Root Container's IWidget. The event will be routed to all applicable containers and widgets for handling. IWidget_HandleEvent() will return TRUE if the event was handled by the UI, FALSE otherwise.
6. The applet can optionally handle the event again. IAPPLET_HandleEvent() must return TRUE if the applet (or any method to which the event handling was delegated) handled the event.

The timely handling of events is crucial to the stability of the system. Failure to return from IAPPLET_HandleEvent() methods can result in watchdog timeouts.



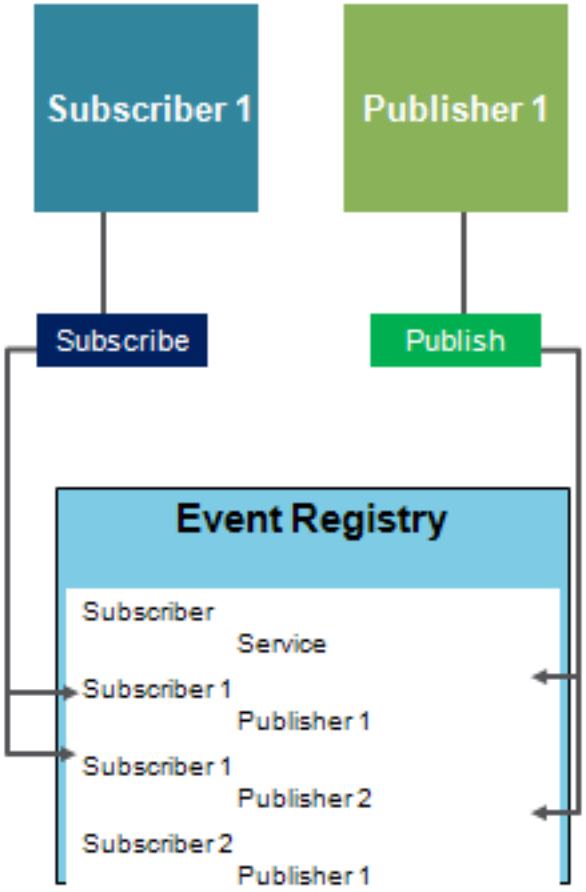
For more event handling information, see the *Application Management Technology Guide for Developers* on the Brew MP website.

Publish and subscribe design pattern

Certain events, such as EVT_NOTIFY, require the applet to register to receive the event. When the event is generated, it is only sent to those clients that have registered to receive the event. This is the Publish and Subscribe design pattern.

The diagram below illustrates the following design pattern:

- Subscriber 1, a client for a service, is an object or interface that requests data to be provided by another service. The data provided is an event code, a notification, or other data.
- Publisher 1 is a service, such as the system, an object, or an interface that generates the event code, notification, or other data.
- A registration database, or system registry, is maintained by the system. This registry records and maps each subscriber to the requested service publisher. Other parameters can be registered to determine how the service is posted.
- The publisher generates a message and sends it to the system registry. The system then dispatches the message to all registered subscribers for that message.



For more information on event handling, see the *Application Management Technology Guide for Developers* on the Brew MP Dev Net.

Event registration

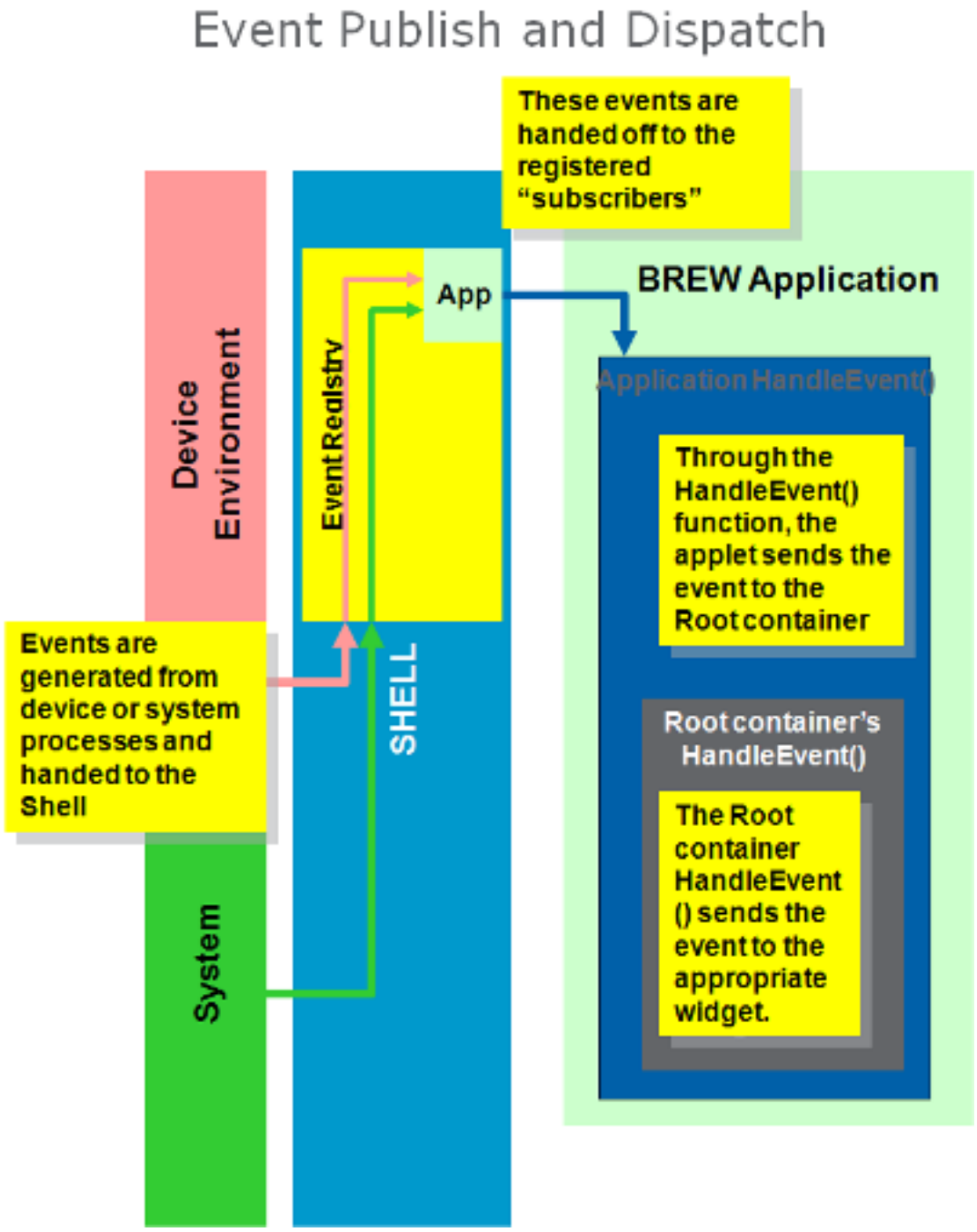
The first step in the design pattern is to subscribe to receive events. The AEE Shell maintains an event registry. Applets register themselves with the shell to receive specific events from specific publishers.

Event publish and dispatch

Events can be generated from several sources, including the device environment (key presses , etc.) or from the system (battery level warning, etc.). Since these services post events without knowledge of which clients are receiving the events, a mechanism is required to send the event to the appropriate subscribers.

The publishing of events from services, the system, or device environment, is handled through the AEE Shell and the event registry. The shell receives the events as native event codes and then posts them to the event registry. The registry then publishes the event to each subscribing service.

When the subscriber receives the event, the HandleEvent() function receives the event as an event code along with other contextual data. The subscriber then processes the event in the application's main event loop.



Key press events

Key press events are generated by keypad use, and are sent to the event handler as EVT_KEY.

When a key is released, the event EVT_KEY_RELEASE is sent to the event handler. An application can determine if a key is being held by receiving notification that the EVT_KEY event is sent, then waiting for the EVT_KEY_RELEASE event. Along with pressing and releasing keys, applications can also respond when a key is pressed and held.

- EVT_KEY is the standard key press event. It is used to receive the action of pressing and releasing a key.
- EVT_KEY_PRESS is a response to a key press event. It is used to receive only the press (down) action of the key.
- EVT_KEY_RELEASE is the response to a key being released. It assumes the key press event has occurred.

Key press parameters include the following:

- wParam indicates which key was pressed.
- dwParam contains the bit modifier flags.

See AEEVCodes.h for more information.

The following is a description of a key press events sequence.

1. EVT_KEY_PRESS is sent when a key is pressed.
2. When a key is held, multiple EVT_KEY events are sent.
3. When a key is released, EVT_KEY is sent first, then EVT_KEY_RELEASE.

The clear key sets wParam equal to AVK_CLR, to move to the previous screen.

Suspend and resume

The current top-visible Brew MP application is suspended when another application needs to become top-visible to have access to the display and the keypad. Examples include:

- Low battery warning
- Incoming phone call
- Incoming non-BREW SMS message

To demonstrate what happens during suspend/resume, consider the case of an incoming call while a Brew MP application is running. EVT_APP_SUSPEND and EVT_APP_RESUME go hand-in-hand, while the events EVT_APP_STOP and EVT_APP_START go hand-in-hand.

1. Brew MP sends the EVT_APP_SUSPEND event to the running application.
2. If the application does not handle the EVT_APP_SUSPEND (i.e., returns FALSE), Brew MP sends EVT_APP_STOP to the application.
3. When the call ends, Brew MP sends EVT_APP_RESUME or EVT_APP_START to the application, depending on whether the EVT_APP_SUSPEND was handled earlier.

When EVT_APP_SUSPEND is received, the following should take place:

- Cancel callback functions and timers.
- Stop animations.
- Release socket connections.
- Unload memory intensive resources.

Note: Each carrier has different guidelines for how an application performs when it is suspended/resumed; refer to carrier guidelines for details.

For more information on suspend and resume, and other application management concepts, see the *Application Management Technology Guide for Developers* on the Brew MP Dev Net.

Signals, callbacks, timers and alarms

Brew MP uses a single-threaded UI model that allows for cooperative multitasking on this thread. Resources are limited on Brew MP devices, so this thread is monitored by an internal watchdog device

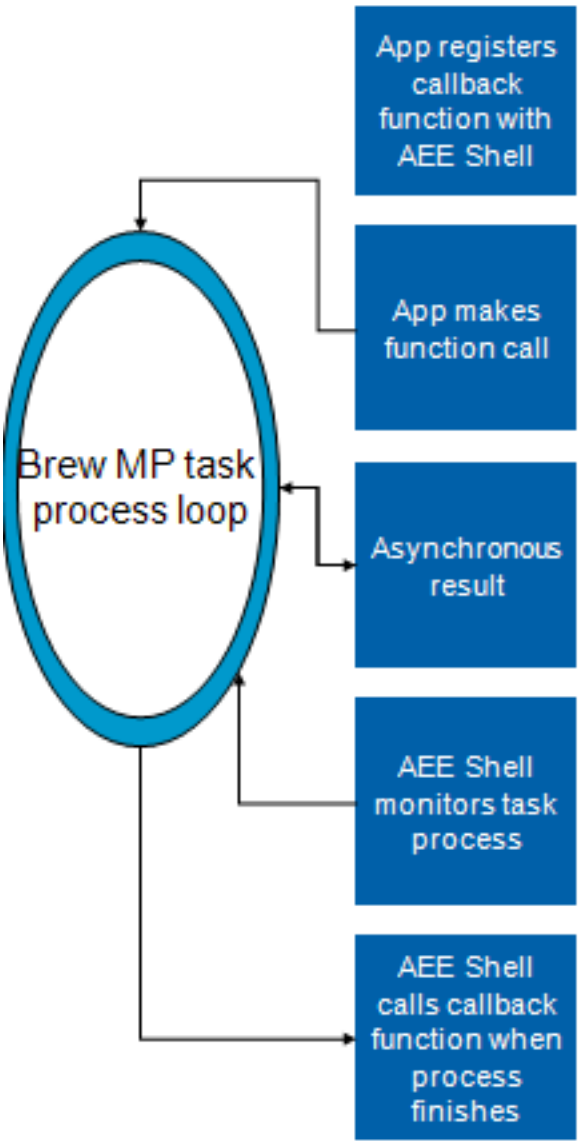
to make sure function calls do not block the thread. You can think of the thread as a dispatch loop, where functions are passed to be processed and on very short intervals the return values are passed back to the calling function. If a function makes a call that takes time to process, the watchdog timer shuts down the application. This typically occurs after approximately 30 seconds. To prevent this blocking situation, signals or callbacks can be used to monitor the dispatch loop for return of the calling function. Applications must use non-blocking calls to avoid locking the device. Signals or callbacks provide a mechanism for processing without blocking calls. In Brew MP there are only separate threads for background thread applications. Foreground UI applications are co-operative threaded.

Signals

Signals are notification objects sent within a process or across process boundaries that provide a method of inter-process communication. For more information on signals, see the *OS Services Technology Guide for Developers* on the Brew MP Dev Net.

Callbacks

A callback is executable code (a function) that is passed as an argument to other code, which allows one software component to call a function defined in another software component. In Brew MP, it is used as a notification mechanism. Instead of blocking the dispatch loop for data to be available, the application can register a callback function and return control back to the dispatcher. When the data become available, the callback function is called and the application can resume its operation. This mechanism is only suitable when it's safe to pass the callback function pointer to the registering function.



Timers

Timers perform an action when a specific amount of time has passed. These time periods are typically short (seconds or milliseconds). Timers are callback-based, and each timer is only triggered once. Users must reset the timer if they want it to repeat. This is usually done inside the callback function itself if a repeating timer is needed. The watchdog timer resets the device if functions don't return in a timely manner.

The AEE Shell's timer facility is used by a currently instantiated application (that is, an application whose reference count is non zero) to perform an action when a specified amount of time has passed. You can use AEE Shell's alarm functions to obtain notification when longer time periods have passed, even when your application is not currently instantiated.

The following steps demonstrate an example of using a timer for animation:

1. Set the timer.

2. When a function is called, draw the graphics, then set another timer.
3. Repeat the behavior for smooth animation.

The same timer and animation routine would automatically work on a faster CPU, without any recoding on your part.

The following are common timer functions:

- ISHELL_SetTimer()
- ISHELL_CancelTimer()
- ISHELL_GetTimerExpiration()

For MOD1 applications, timer functionality is found in ISysTimer.

Alarms

Alarms allow for notification when time reaches a specific value. If an application is not running, Brew MP starts it, then sends it the alarm event. Alarams are typically used when the time of notification is in the distant future, such as calendar alarms that can be used to alert the user when the time of a calendar appointment is about to be reached. Each alarm only triggers once.

The AEE Shell's alarm functions enable an application to be notified when the current time reaches a specified value. Unlike timers, which can only be active while your application is running, you can receive notification that an alarm has expired even when your application is not running.

Note that like timers, alarms do not repeat.

The following are common alarm functions:

- ISHELL_SetAlarm()
- ISHELL_CancelAlarm()

Using callbacks with timers

1. Create a function prototype for your callback function.
2. Call ISHELL_SetTimer() with the address of the callback function and a pointer to an application-specific data structure. For example:

```
ISHELL_SetTimer (pMe -> IShell , TIMER_VAL, (PFNNOTIFY) MyFunc , pMe);
```

3. When the timer expires, the shell calls the callback function.

Canceling timers

An individual timer can be cancelled with ISHELL_CancelTimer, as follows:

```
ISHELL_CancelTimer (pMe -> pIShell, (PFNNOTIFY) MyFunc, pMe);
```

All timers with the same data pointer can be canceled by passing NULL as the function pointer, as follows:

```
ISHELL_CancelTimer (pMe -> pIShell, NULL, pMe);
```

AEECallback

AEECallback is recommended over function pointers alone. AEECallback is a structure that contains a function pointer, a data pointer, and other bookkeeping data and this structure can be passed quite easily. Much of the bookkeeping data is instantiated by Brew MP and should not be directly accessed by your application.

To initialize the AEECallback, call the CALLBACK_Init() method, passing in a pointer to the structure and a pointer to your callback function, as shown in the code below:

```
CALLBACK_Init (& pMe -> myCallBack, (PFNNOTIFY) MyFunc, pMe );
```

Note that this AEECallBack structure should remain valid throughout the asynchronous request to ensure that the function pointer and data are available. The CALLBACK_Cancel() macro can be used to cancel the callback, such as in the case of suspend events. This macro takes a pointer to an AEECallback structure.

The following code fragment shows the initialization of a callback structure AEECallback named myCallBack, which is a member of the applet struct. The structure is initialized using the CALLBACK_INIT() function, marking MyFunc as the function to call when task processing is returned to the application. When the timer is set using ISHELL_SetTimerEx(), the callback is set as the third parameter.

```
//In app struct
AEECallback myCallBack;

// MyFunc is a function defined in your application
// pMe is the pointer to your application struct

CALLBACK_Init(&pMe->myCallBack, (PFNNOTIFY)MyFunc, pMe);
ISHELL_SetTimerEx(pMe->pIShell, TIMER_VAL, &pMe->myCallBack);
```

For more information, see AEECallback in the *C/C++ API Reference*.

Notifications

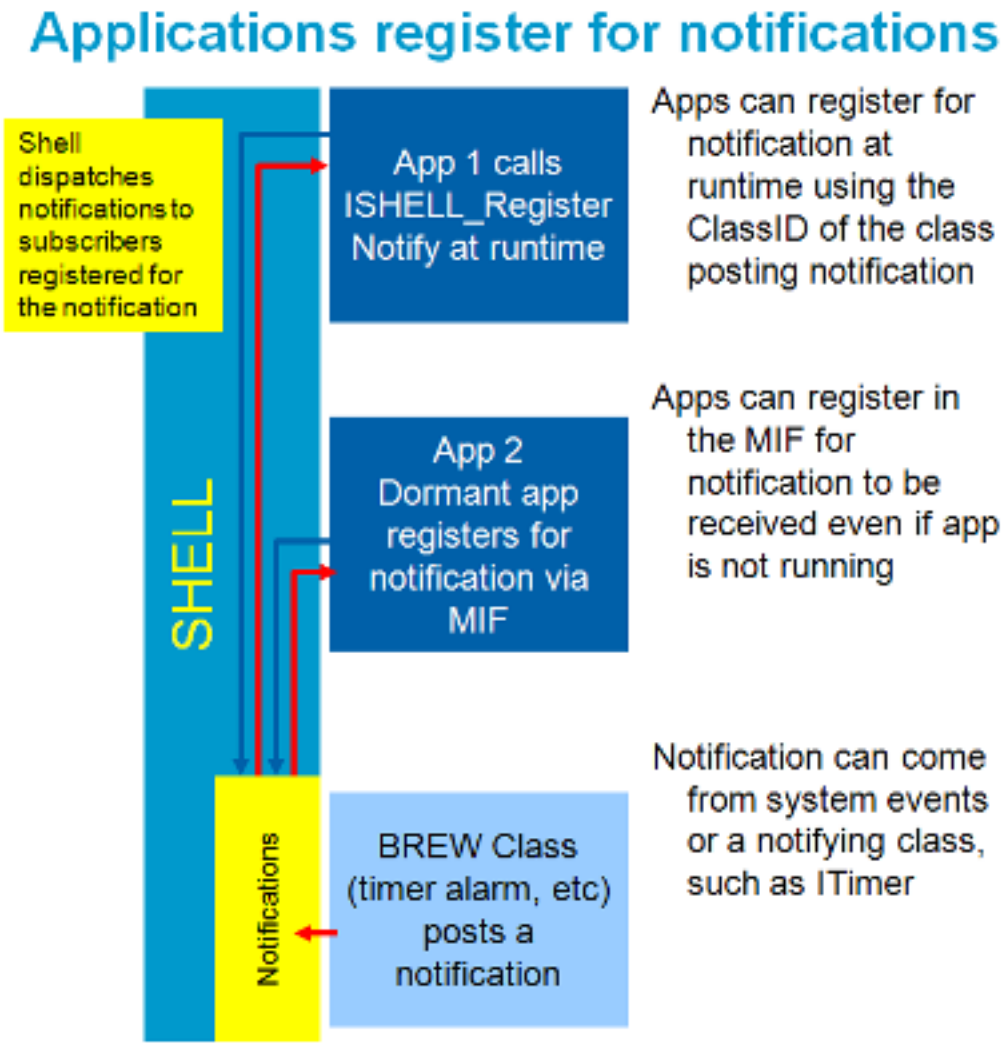
IShell's notification mechanism allows a Brew MP class to notify other classes that certain events have occurred. To receive a notification, a class must register its interest with the AEE Shell, specifying the ClassID of the notifier class and the events for which notification is desired. When an event requiring a notification occurs, the notifier class calls ISHELL_Notify( ), sending a notification to each class that has registered to be notified of the occurrence of that event.

The AEE Shell provides two ways for a class to register for notification of an event:

- You can register by specifying information about the notification in your application's CIF/MIF file through the Notifier primitive (see the *Resource File and Markup Reference*). This method of registering is used by applications that must be notified of events even when they are not running. Brew MP maintains a list of notifications based on the information in the MIF. Brew MP can send a notification to an applet that isn't running, wake up the applet, and the notification is received by the application's HandleEvent function. One example is a call-logging application that receives notification of each incoming and outgoing call; such an application would need to process notifications even while the user was not running the application to display the call log.
- If notification is required only at certain times while your application is running, you can call ISHELL_RegisterNotify( ) to initiate event notification. For example, a game application might display a message when an incoming text message arrives that would allow the user to open and read the message or continue playing the game. This application requires notification of incoming text messages only while the user is actually playing the game, so it would call ISHELL_RegisterNotify( ) when the user starts to play the game.

The following diagram shows how applications register for notifications.

Notifications



Receiving Notifications

To receive a notification, the application must register for notifications. This can be done in one of two ways:

- ISHELL_RegisterNotify() can be called during run-time to register for a notification from a given applet (ClassID). Notifications can only be received after this call is made and registered with the BREW Shell.
- The notifications to receive can be listed in the application MIF. This enables receipt of notifications even when the application isn't running. This approach can be used to notify when the BREW Shell is first initialized by listening to a notification from AEECLSID_SHELL called NMASK_SHELL_INIT.

For example, to register for FLIP, KEYGUARD, and SCR_ROTATE notifications in the CIF, add the following Notifier primitives to the CIF:

```
Notifier {
   clstype = AEECLSID_DEVICENOTIFIER,
   clsnotify = AEECLSID_MyApp,
   mask = NMASK_DEVICENOTIFIER_FLIP,
}
Notifier {
   clstype = AEECLSID_DEVICENOTIFIER,
   clsnotify = AEECLSID_MyApp,
   mask = NMASK_DEVICENOTIFIER_KEYGUARD,
}

Notifier {
   clstype = AEECLSID_DEVICENOTIFIER,
   clsnotify = AEECLSID_MyApp,
   mask = NMASK_DEVICENOTIFIER_SCR_ROTATE,
}
```

To register for FLIP, KEYGUARD, and SCR_ROTATE notifications using ISHELL_RegisterNotify(), add the following code to the application:

```
IShell_RegisterNotify(piShell, AEECLSID_MyApp, AEECLSID_DEVICENOTIFIER,
         NMASK_DEVICENOTIFIER_FLIP | NMASK_DEVICENOTIFIER_KEYGUARD |
         NMASK_DEVICENOTIFIER_SCR_ROTATE);
```

When an application registers for notifications, it receives the EVT_NOTIFY event when one of the specified events occurs. The application can use the notification masks to determine which event occurred. For example:

```
static boolean c_app_HandleEvent(c_app* pMe, AEEEvent eCode,
            uint16 wParam, uint32 dwParam)
{

   switch (eCode){
      case EVT_NOTIFY:
      {
         if(dwParam){
            AEENotify *pNotify = (AEENotify *) dwParam;
            AEEDeviceNotify * pDevNotify = NULL;

            //first check if it is a device notification
            if ( (AEECLSID_DEVICENOTIFIER == pNotify->cls){
               // check if it is SCR_ROTATE
               if(NMASK_DEVICENOTIFIER_SCR_ROTATE | pNotify->dwMask){
                  // pDevNotify->wParam will the the same wParam that
                  // is sent with EVT_SCR_ROTATE
                  // pDevNotify->dwParam will the same dwParam that
                  //is sent with EVT_SCR_ROTATE

                  pDevNotify = (AEEDeviceNotify *) pNotify->pData;
               }
               // check if it is FLIP
               if(NMASK_DEVICENOTIFIER_FLIP | pNotify->dwMask){
                  // pDevNotify->wParam will the the same wParam that
                  // is sent with EVT_FLIP
                  // pDevNotify->dwParam will the same dwParam that is
                  // sent with EVT_FLIP

                  pDevNotify = (AEEDeviceNotify *) pNotify->pData;

               }
            // continue to check HEADSET, KEYGUARD if needed…
```

```
            }

         }
         break;
      }
   }
}
```

Sending Notifications

To send a notification, call ISHELL_Notify(). This results in all the applets registered for this notification receiving an event called EVT_NOTIFY, along with the data sent with ISHELL_Notify().

The *C/C++ API Reference* contains more information on the functions mentioned, as well as related functions for communicating between applets such as ISHELL_BrowseURL(), ISHELL_PostURL(), and ISHELL_SendEvent().

For more information on notifications, see the *Application Management Technology Guide for developers*, on the Brew MP Dev Net.

Implementing classes

This section provides information on implementing the following classes:

- applet class on page 47
- in-process class on page 47

Implementation of service classes is discussed in the *Programming Model for Manufacturers*.

Implementing an applet class

To implement an applet class, do the following:

1. Declare the applet class in the CIF for the module.

   Each applet class must have a unique class ID (i.e. AEECLSID_XXXX). An applet class can also specify its privileges (AEEPRIVID_XXXX) in the CIF file.
2. Implement the applet class in C/C++.
3. Build the applet class, as follows:
   a. Compile the CIF and any CAR files to create the MIF and BAR files.
   b. Compile the C/C++ source code, using an arm compiler, to generate the module for the applet class.

For an example implementation of an applet class, see the c_basicapp example in the Brew MP Sample Code.

The Brew MP IDE plugin for Visual Studio or Eclipse provide a wizard to facilitate the development of applet classes.

Implementing an in-process class

To implement an in-process class, do the following:

1. Declare the in-process class in the CIF for the module.

   Each in-process class must have a unique class ID (AEECLSID_XXXX).
2. Implement the in-process class in C/C++.

3. Build the in-process class, as follows:
   a. Compile the CIF and any CAR files to create the MIF and BAR files.
   b. Compile the C/C++ source code, using an arm compiler, to generate the module for the in-process class.

For an example implementation of an in-process class, see the c_basicext example in the Brew MP Sample Code.

The Brew MP IDE plugin for Visual Studio or Eclipse provides a wizard to facilitate the development of in-process classes.

Key APIs

This section provides an overview of some of the key Brew MP APIs and includes the following topics:

Purpose of the APIs

- To standardize the programming environment for portability to multiple wireless products.
- To minimize the use of system resources.
- To shield application developers from having to deal directly with device drivers, telephony, etc.

Brew MP C/C++ API Reference

The *C/C++ API Reference* provides comprehensive information about the use of all Brew MP interfaces and helper functions, along with the data structures and constants that accompany them. Each interface is explained in an overview, followed by detailed information for each interface function. Related topics are hyperlinked for easy navigation to relevant information.

The C/C++ API Reference is not tutorial in nature. How-Tos and sample code can be found on the Brew MP Dev Net.

IModule and IMod

IModule is used with the MOD/DLL format, and IMod is used with the MOD1/DLL1 format.

IModule

The IModule interface provides a mechanism for controlling access to a group of associated classes (applets and in-process classes) in a MOD file. The IModule interface allows modules to expose a wide variety of classes without fixed entry points.

IModule can be leveraged to implement a singleton contract. The IModule interface is a singleton enforced by Brew MP at creation time.

When Brew MP loads a module, the module is represented by a unique instance of the IModule class. During execution, IModule creates instances of the classes it contains when requested by applications. After all instances of these classes in the module are released, the module is released, freeing the memory it used to contain its code and data. Brew MP only loads and creates a module once.

AEEModGen.c is the implementation for the IModule interface. AEEModGen.c is a helper file provided by the Brew MP Plugin Wizard, containing reference source code for modules. IModule is the pointer to the module (DLL or MOD file), and deals with loading the entry point. It contains code that enables the MOD file to be loaded, and exposes the single CreateInstance entry point to it.

AEEMod_Load() is the entry point for the module in MOD format and is implemented in AEEModGen.c. AEEModGen.c also implements four methods of IModule:

- CreateInstance: Brew MP invokes this method when it needs an instance of a class provided by the module.
- FreeResources: this method frees additional resources consumed by the module prior to its destruction.
- AddRef: this method increments the module's reference count.
- Release: this method decrements the module's reference count and cleans up after the module when its reference count reaches 0.

For more information on IModule, see the Brew MP *C/C++ API Reference* on the Brew MP Dev Net.

AEEClsCreateInstance() is the entry point for all classes inside the module and needs to be implemented in source file of the classes. It is invoked by the AEEMod_CreateInstance() in AEEModGen.c file when a class is being instantiated.

When the component infrastructure receives a request to instantiate an object via IEnv_CreateInstance(), it performs the following actions:

1. Searches the module database (the set of all MIF files in the system) for a module that implements the class.
2. Loads and links the module's code image in memory.
3. Calls the entry function, AEEMod_Load(), in the newly loaded module to instantiate the component.
4. Calls AEEMod_CreateInstance() on the resulting IModule, which calls AEEClsCreateInstance() to instantiate the requested class. When idle modules (modules that have no references to their IEnv) are released, AEEMod_FreeResources() is called before the module's code is released.

For more information on IModule, see the Brew MP *C/C++ API Reference* on the Brew MP Dev Net.

IMod

IMod is used to instantiate classes implemented by a module (DLL1 or MOD1 file). Each module exports one entry function, IMod_New, executed when the module is loaded. This function must return an instance of IMod, which is then used to instantiate classes implemented in that module.

Each module must provide implementation of IMod_New(). Modules typically do this by linking with a1mod and building with code auto-generated by the CIF compiler (cifc.exe), which contains the implementation for IMod_New.

When the component infrastructure receives a request to instantiate an object via IShell_CreateInstance() or IEnv_CreateInstance(), it performs the following actions:

1. Searches the module database (the set of all MIF files in the system) for a module that implements the class.
2. Loads and links the module's code image in memory.
3. Calls the entry function, IMod_New(), in the newly loaded module to instantiate the component.
4. Calls IMod_Init() on the resulting IMod.
5. Calls IMod_CreateInstance() to instantiate the requested class. When idle modules (modules that have no references to their IEnv) are released, IMod_Exit() is called before the module's code is released.

For more information on IMod, see the *OS Services Technology Guide for Developers* on the Brew MP Dev Net.

IShell and IEnv

For every loaded and initialized Brew MP module, an IShell or IEnv pointer is passed to module code when its classes are instantiated.

Class inheritance from BREW

Brew MP incorporates the BREW Application Framework, which provides a runtime environment for Brew MP applications. Applets run in Brew MP within the environment created by the application framework. Each environment scope is identified by an instance of the shell. The application framework provides applications with an instance of the shell interface (IShell). Shell coordinates applets at runtime by providing them with a method to create instances of classes by ID. The application framework also serves applets with a set of utilities in AEEStdLib.h.

Most IShell-based classes in Brew MP were inherited from the BREW platform. In BREW there is no clear separation between application framework and platform services. There is only one shared execution environment and memory domain for the platform services. Some of these classes also made the following assumptions:

- The main goal for these programs is to provide API services for direct consumption by applets or indirect consumption by another IShell-based program. Some of these programs are strictly for direct use by applets, and the implementation requires that the user of the program be capable of receiving events on IApplet_HandleEvent(). Examples are deprecated controls such as text, menu, etc.
- Objects in these programs were created by the shell instance, so they always had access to pointer to IShell. The implementation had heavy dependency on the shell, application local storage (or application globals), and interfaces in AEEStdLib.h.
- They typically run in the single designated BREW thread in the system.

A few other program types belong to one of the above categories. They may also support a contract needed by a particular framework. Some popular ones are the following:

- Applets are IShell-based programs that implement the IApplet interface. These programs typically incorporate a user interface and are visible to the user in the application launcher.
- Notifiers are the IShell-based programs that implement the INotifier interface. Applets typically subscribe to notifiers to receive notifications. The BREW application framework mediates the subscription and notification operations between notifiers and applets.

IShell

An IShell object is the first object provided to each class when created in a MOD file. All classes created in MOD are by default IShell-based classes. A class (or any class it uses) that has access to an IShell object is instantiated and used inside the BREW Shell, and has an IShell dependency. The IShell object is always available, exposes objects and services provided by the BREW Shell, and can be used to discover and create other objects available in the system. It can discover and create any other object in the system via IShell and IEnv.

An IShell-based class is defined as a class that is given the IShell object when instantiated. This can be observed by examining whether an IShell pointer is passed to the class by the system via the constructor or the new function of your class. For example:

```
int AEEClsCreateInstance(AEECLSID ClsId, IShell *pIShell,
```

```
                IModule *po, void **ppObj)
```

IShell-based classes exhibit the following characteristics:

- Single-threaded implementation
- BREW Shell is used, with interface IShell
- Only used within the execution environment provided by the BREW application framework
- Can't be used as a service class

In Brew MP, applets can be created to run in another process (an isolated protection domain), thus running in a separate thread. Applets hosted in a separate process also have an instance of shell. An IEnv object can be derived via IShell, but IShell can't be multi-threaded, since it resides in the single-threaded BREW Shell.

All pre-existing applets, and any new ones that do not specify that they should run in a separate process, will continue to work in the legacy shell space. Legacy shell instances are fully backward compatible for applets created for BREW 3.1.2 and later BREW versions.

Applets with an affinity to the legacy shell are referred to as legacy applets. To maintain feature support and backward compatibility for legacy applets, lthe legacy shell is hosted in the kernel process in Brew MP. See kernel process on page 18 for more information.

IEnv

An IEnv object is the first object provided to each class created in a MOD1 file. An IEnv object exposes services provided by Env, such as class discovery, instantiation, and memory allocation.

An IEnv-based class is defined as a class that was given an IEnv object when instantiated. This can be observed by examining whether an IEnv pointer is passed to the class by the system via the constructor or the new function of your class. For example:

```
int c_basicmod1app_New( IEnv* piEnv, AEECLSID cls, void** ppif)
```

Each object in Brew MP resides in an IEnv and has access to an IEnv object. If it has no IShell dependency, it can be instantiated and used inside as well as outside the BREW Shell. If instantiated inside the BREW Shell, it can discover and create any other object in the system via IShell and IEnv. An IShell object can be derived via IEnv.

If instantiated outside the BREW Shell, the IEnv object can be used to discover and create other IEnv-based objects that don't have an IShell dependency. It can be multi-threaded when instantiated outside the BREW Shell. Some Brew MP APIs and all the classes in MOD1 are IEnv classes.

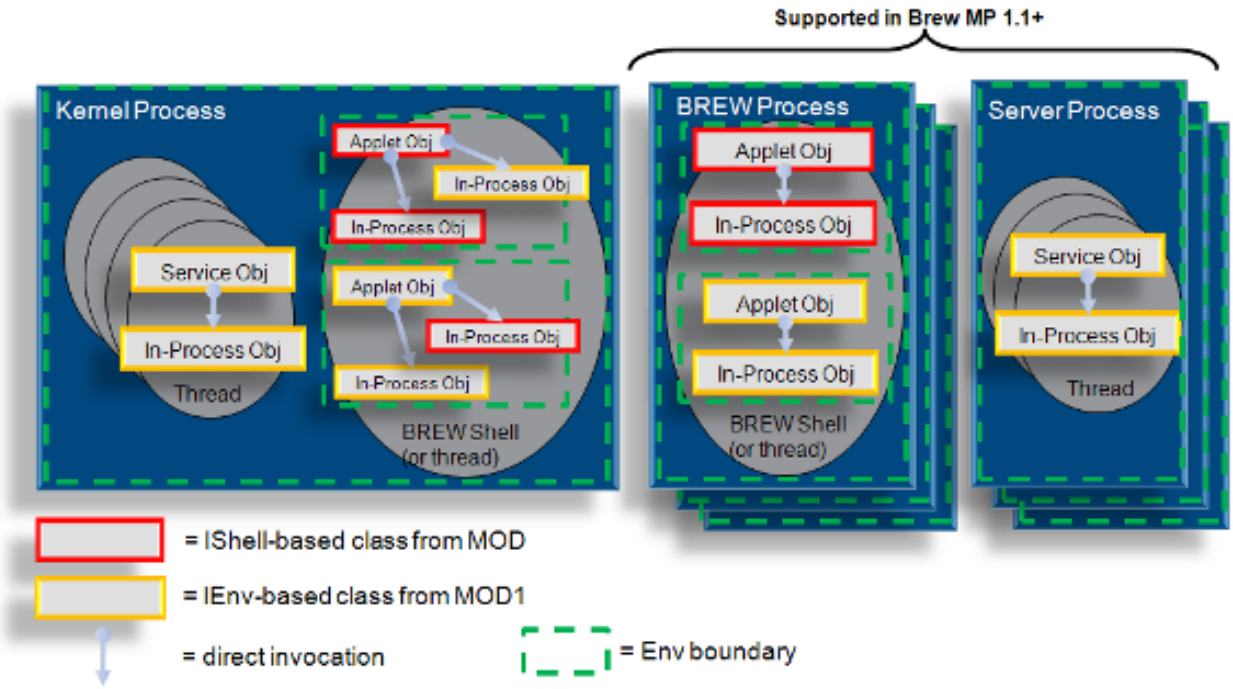
IEnv-based classes that have IShell dependency have all of the same characteristics of IShell-based classes discussed previously. IEnv-based classes without IShell dependency exhibit the following characteristics:

- Can be a service class
- Can contain multi-threaded implementation
- Free of BREW Shell
- Can be used inside or outside the execution environment provided by the BREW application framework.

For more information on IEnv, see the *Memory and Heap Technology Guide for Developers*, on the Brew MP Dev Net.

IShell and IEnv

The following diagram shows how both IShell and IEnv manifest in runtime memory.



Widgets and IDisplay

It is recommended you use Widgets to create user interfaces, instead of using IDisplay. You can also use windows, which provide additional functionality.

For more information on Widgets and windows, see the*Widgets Technology Guide* and *Window Manager Technology Guide* on the Brew MP Dev Net.

ISettings

ISettings allows the storage of settings in a key value format, stores application and user data, specifies permissions, and restricts access by making certain settings private or public.

ISettings allows applets to get and set named keys with string values, supports heirarchical key structures, allows enumerating over trees of keys, supports notifications when values change, and is remotable.

Public settings

Public settings are made available to the rest of the system through a global settings registry. This settings registry is provided by a singleton service.

The registry uses a URI-based settings tree (</path/to/a/setting>). This settings registry is made up of smaller sub-registries provided by any number of modules that plug in to the top-level registry. Each sub-registry owns a specific part of the tree, manages its own data store, and can manage its own privileges.

The settings registry allows various types of data storage and supports change notifications across applications.

Private settings

Private settings are available to an application or component on a per-instance basis. These settings aren't exposed to any other component in the system. Private settings are created using a factory and require no modification of the MIF. The factory creates the desired ISettings object at run-time.

.ini file-based factories create settings that live in .ini files and persist in the EFS. Heap-based factories create settings in the heap, which do not persist across power cycles.

I want to...	Setting type	Storage type	Settings Registration in MIF	Settings ACLs in MIF	Additional notes
...expose my app's .ini file-based settings to the system	Public	.ini	Yes	Optional	Must provide .ini file
...expose my app's heap-based settings to the system	Public	heap	Yes	Optional	
...expose my custom settings to the system	Public	custom	Yes	No	Must provide custom ISettings implementation
...store some persistent data in my component	Private	.ini	No	No	
...organize some non-persistent state information in my component	Private	heap	No	No	
...share some non-persistent state information between component instances	Public	heap	Yes	Optional	
...get notifications on a private heap store	Private	custom	No	No	Must create a custom ISettings class that internally uses the heap factory. All methods are delegated to the object from

					the factory, but OnChange must be implemented and Set, Delete, and Reset must trigger a notification.

Adding a public store

The following sections cover public .ini file-based, public heap based, and public custom settings store.

Type: public .ini file-based

Perform the following steps to add and access a public .ini file-based settings store to your component:

1. Copy the following into a file called mysettings.ini and place it into your component's module directory in EFS.

```
[section1]
setting1=value1
```

2. Register the store with the system by adding the following to your component's CIF file.

```
local s = require 'SettingsCIFHelpers'
-- register my settings at "/myApp/myIniSettings/..."
s:RegisterIniFile {
   owner = 0x12345678, -- class id of my component
   key = "/myApp/myIniSettings",
   file ="mysettings.ini",
   acls = { ... } -- optional
}
```

3. ACLs to allow other applications access to your settings are optional in the CIF. For example:

```
acls = {
   { -- grant everyone read access to my settings but write access
     -- to only those modules belonging to the 0xdeadd00d group
      {
         groups = {0},
         perms = "r/r",
      },
      {
         groups = {0xdeadd00d},
         perms = "rw/rw",
      },
      path = "/myApp/myIniSettings"
   },
}
```

4. The following code accesses your setting:

```
{
   ISettings *piSettings = NULL;
   if (SUCCESS == IEnv_CreateInstance(piEnv, AEECLSID_SettingsReg,
          (void**)&piSettings)) {
      char outbuf[32];
      int result;
      result = ISettings_Get( piSettings,
              "/myApp/myIniSettings/section1/setting1",
```

```
                outbuf, sizeof(buf), NULL
      );
      if (SUCCESS == result) {
         // outbuf will contain "value1"
      }
      ISettings_Release(piSettings);
      pSettings = NULL;
   }
}
```

Type: public heap-based

Adding and accessing a public heap-based settings store to your component is similar to the .ini file-based store. The main difference is the absence of the .ini file. Instead, heap-based settings require a quota value that determines the maximum amount of heap that may be used by the store.

1. Register the store with the system by adding the following to your component's CIF file:

```
local s = require 'SettingsCIFHelpers'

-- register my settings at
"/myApp/myHeapSettings/..."
s:RegisterHeap {
   owner = 0x12345678, --class id of my component
   key = "/myApp/myHeapSettings",
   quota = 0x1000, acls = { ... }
}
```

2. ACLs to allow other applications access to your settings are optional in the CIF. For example:

```
acls = {
   { -- grant everyone read access to my settings but write access
     --  to only those modules belonging to the 0xdeadd00d group
      {
         groups = {0},
         perms = "r/r",
      },
      {
         groups = {0xdeadd00d},
         perms = "rw/rw",
      },
      path = "/myApp/myHeapSettings"
    },
}
```

3. The following code accesses your setting. Note that a heap-based setting does not exist until ISettings_Set() is called on it.

```
{
   ISettings *piSettings = NULL;
   if (SUCCESS == IEnv_CreateInstance(piEnv, AEECLSID_SettingsReg,
                  (void**)&piSettings)) {
      char outbuf[32];
      int result;
      (void) ISettings_Set(piSettings, "/myApp/myIniSettings/foo",
                "bar");
      result = ISettings_Get( piSettings, "/myApp/myIniSettings/foo",
                outbuf, sizeof(buf), NULL
      );
      if (SUCCESS == result) {
         // outbuf will contain "bar"
      }
      ISettings_Release(piSettings);
      pSettings = NULL;
   }
}
```

Type: public custom

Perform the following steps to add a custom ISettings implementation into the settings registry:

1. Write a component that implements ISettings.
2. Register the component with the system by adding the following code to the component's CIF file:

```
local s = require 'SettingsCIFHelpers'

s:RegisterClass {
class = 0xdeadbeef,
key = "/myApp/myCustomSettings",
}
```

3. Use the following code to access your setting. Note that any ISettings operations performed on the registry class are delegated to the custom class.

```
{
   ISettings *piSettings = NULL;
   if (SUCCESS == IEnv_CreateInstance(piEnv, AEECLSID_SettingsReg,
       (void**) &piSettings)) {
      int nChildren = 0; int result;
      result = ISettings_GetNumChildren(piSettings,
               "/myApp/myCustomSettings",
               &nChildren);
      if (SUCCESS == result) {
         // do something
      }
   ISettings_Release(piSettings);
   pSettings = NULL;
   }
}
```

Creating a private store

The following sections cover private .ini file-based, private heap-based, and private custom settings stores.

Type: private .ini file-based

Perform the following steps to access a private .ini file-based settings store from your component:

1. Copy the following into a file called mysettings.ini and place it into your component's module directory in EFS.

```
[section1]
setting1=value1
```

2. Use the following code to access your setting.
   Note that unlike public settings, access to the private store does not require the prefix "/myApp/myIniSettings".

```
{
   ISettingsStoreFactory *piSSF = NULL;
   if (SUCCESS == IEnv_CreateInstance(piEnv,
              AEECLSID_SettingsIniFactory, (void**)&piSSF)) {
      ISettings *piSettings = NULL;
      int result; result = ISettingsStoreFactory_Create( piSSF,
                        "owner=0x12345678;path=mysettings.ini",
                        &piSettings );
      if(SUCCESS == result) {
         char outbuf[32];
         result = ISettings_Get( piSettings, "section1/setting1",
```

```
                              outbuf, sizeof(buf), NULL );
         if(SUCCESS == result) {
            // outbuf will contain "value1"
         }
         ISettings_Release(piSettings);
         pSettings = NULL;
      }
      ISettingsStoreFactory_Release(piSSF);
      piSSF = NULL;
   }
}
```

Type: private heap-based

Accessing a private heap-based settings store from your component is similar to the .ini file-based store. The main difference is the absence of the .ini file. Instead, heap-based settings require a quota value that determines the maximum amount of heap that may be used by the store.

Use the following code to access your setting. Note that a heap-based setting does not exist until ISettings_Set() is called on it.

```
{
   ISettingsStoreFactory *piSSF = NULL;
   if (SUCCESS == IEnv_CreateInstance(piEnv, AEECLSID_SettingsHeapFactory, (void**)
         &piSSF)) {
      ISettings *piSettings = NULL;
      int result;
      result = ISettingsStoreFactory_Create( piSSF,
                          "quota=0x1000", &piSettings );
      if (SUCCESS == result) {
         char outbuf[32];
         (void)ISettings_Set(piSettings, "foo/bar", "Hello world");
         result = ISettings_Get(piSettings, "foo/bar", outbuf,
                    sizeof(buf), NULL);
         if (SUCCESS == result) {
         // outbuf will contain "Hello world" }
         ISettings_Release(piSettings); pSettings = NULL;
      }
   ISettingsStoreFactory_Release(piSSF);
   piSSF = NULL;
   }
}
```

Type: private custom

Perform the following to access a custom ISettings implementation:

1. Write a component that implements ISettings.
2. Use the following code to access your setting:

```
{ ISettings *piSettings = NULL;
if (SUCCESS == IEnv_CreateInstance(piEnv, <classid>, (void**)
                              &piSettings)) {
   int nChildren = 0;
   int result;
   result = ISettings_GetNumChildren(piSettings, "/path/to/my/settings",
                              &nChildren);
   if (SUCCESS == result) {
      // do something
   }
   ISettings_Release(piSettings);
   pSettings = NULL; } }
```

For more information about ISettings, refer to the *Settings Technology Guide for Developers* on the Brew MP Dev Net.

IApplet

IApplet is the base interface for all Brew MP applets. IApplet handles applet-specific functions, and is only required if you are exporting an applet from your module.

Applet classes are event-driven programs. Brew MP invokes IApplet_HandleEvent() with events and event data. The applet class implements logic to handle those events.

The implementation for the IApplet interface is AEEAppGen.c, which includes information in addition to AEEModGen.c required to create an applet. It encapsulates the basic functionality of an applet, mostly message handling. AEEAppGen.c is only provided for MOD applets, not for MOD1 applets.

For more information, see the Brew MP *C/C++ API Reference* on the Brew MP Dev Net.

Brew MP application files

This topic provides an overview of the Brew MP application files and how they work together. The file types are discussed in more detail in the following sections.

A Brew MP application or extension consists of the following required and optional files.

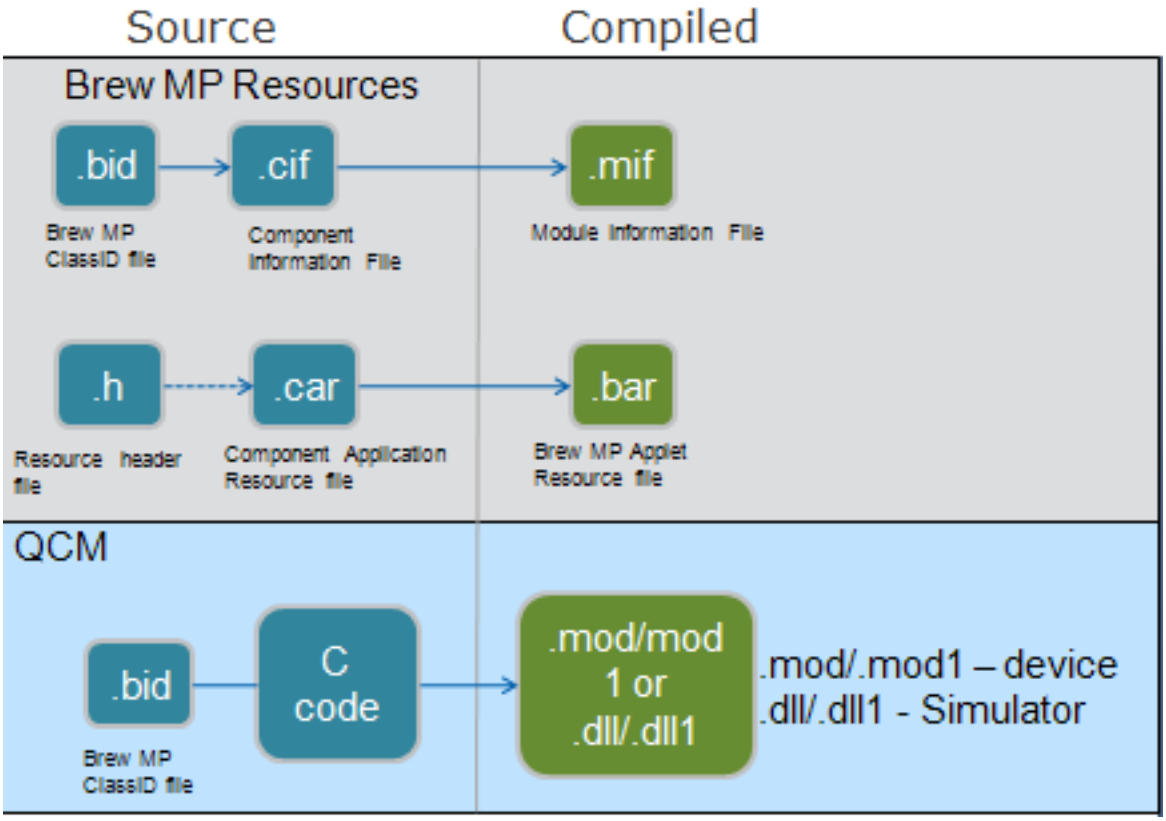
Required file types

- Module Information File (MIF), which provides the application entry point for your application. The MIF is an application descriptor, providing information about external libraries that the application needs for execution.
- Signature file (SIG), which is only required for the dynamic modules on a device. All Brew MP components are digitally signed. The SIG file is stored in the component's home directory; the signature spans at least the module's executable and MIF file.
- DLL or DLL1 for Simulator, built in your development environment.
- The Brew MP module binary file (MOD or MOD1), for devices. The MOD or MOD1 is the binary executable for a Brew MP component. It is digitally signed and stored in the component's home directory. Applications generally use the MOD module format while system components use the MOD1 format.

Optional file types

- One or more BREW Application Resource (BAR) files. The BAR is a compiled, binary file that contains resource information. These resources are localizable strings in menu controls, regional images, or any resource that may change based on handset, language, or region.
- Any number of user files.

The following diagram shows how parts of a Brew MP application are built. An application MIF file, resource file, and application binary are created, then tested on the Brew MP Simulator. The BID files containing Brew MP ClassIDs are also included by the CIF.



Module storage

Each Brew MP component is stored in a dedicated directory in the file system rooted in either the "sys" or "user" directory. The component's home directory is the default location for all files created or accessed by the component.

Application directory structure

Brew MP introduces a single-directory module format. Legacy BREW separated MIF and MOD files. Brew MP introduces collections of files; a collection being a group of modules arranged in a similar way. The application MIF is now in the same path as the rest of the application files. Some standard Brew MP collections are pre-defined, such as those in the following table.

Name	Location	Module Type	Description
BDS Mods	fs:/mif and fs:/mod	MIF and MOD	Backwards compatibility with legacy BREW, Applications under development and/or test. Used for BDS application downloads
User MODs	fs:/UserMods	ONEDIRMOD	New collection for user-installed modules. Not

Name	Location	Module Type	Description
			managed by a download system.
System MODs	fs:/sys/mod	ONEDIRMOD	For system modules such as those that make up Brew MP itself, and those added by OEMs

Application directory

The application directory is defined as a parent directory for Brew MP applications, with each application stored in a subdirectory under this parent. Individual applications reside in subdirectories that include supporting files, such as text files, images, or data files for the application. The binary image for an application is a DLL/DLL1 file for use in the Simulator, and in a MOD/MOD1 file on the device. The application and its directory are required to have the same name. Note that the file system is case-sensitive and all file names specified with the "fs:" qualifier are treated as case-sensitive, but file names not specified with the "fs:" qualifier are treated as case-insensitive. Brew MP converts case-insensitive file names to lowercase.

A MIF contains the application's icons and therefore is best suited for only one color depth. When working with devices that don't all have the same color depth, it's usually easier to switch between different copies of the MIF than to keep modifying a single MIF. The MIF filename must match the name of the application module and its directory.

Unique IDs (BID)

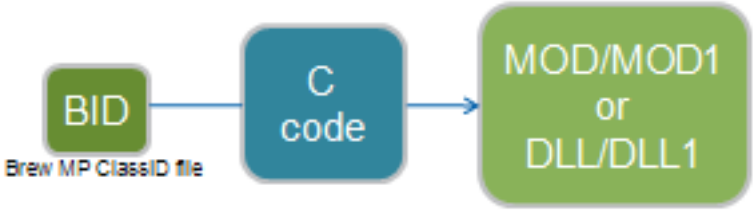
In Brew MP, a unique ID is a 32-bit globally unique unsigned integer.

The unique ID is stored in a Brew MP ID (BID) file, which is a header file with a single define for the ID number. There are three types of unique IDs that are common in Brew MP.

- An Interface ID's friendly name starts with AEEIID. It is used to uniquely identify an interface, and is defined in the interface header.

```
#define AEEIID_IHFont 0x0102ef8e
```

- An ID's friendly name beginning with AEECLSID, is either a ClassID, used to identify a regular class, an Applet ID to identify an applet class, or a Service ID to identify a service class. These IDs should be defined in the BID files. The ClassID is declared through a #define statement matching it to a ClassID name used in application startup, and the ClassID is embedded in the MIF. A #include is used to embed the ClassID into the executable when the application is compiled.




```
#define AEECLSID_FILEMGR 0x01001003
```

- A privilege ID name starts with AEEPRIVID. It is also defined in the BID file and uniquely identifies a privilege to access one or a group of resources. A Group ID is a legacy concept from BREW that is essentially the same thing.

```
#define AEEPRIVID_DISPSETTINGS 0x0103081d
```

All production Brew MP applications require a unique ID. These 32-bit hexadecimal numbers are provided by the BREW ClassID Generator available on the Brew MP Dev Net. The ClassID Generator provides a BREW ClassID (BID) file containing the ClassID.

Trial IDs (local IDs) can be generated locally via the Resource Manager for use with the Brew MP Simulator. The local IDs must be changed to commercially-generated IDs using the ClassID Generator prior to testing the application on a device. The ClassID for an application is stored in an external BID resource file as well as embedded in the MIF.

A bridge exists between the icon that the user selects in some type of application manager and the application that should launch through the embedded ClassID. When the application is installed, its ClassID is registered with Brew MP at runtime and the icon is made available. When the user selects the icon from the Simulator or the device, the runtime environment matches the ClassID from the MIF to the application and launches the application. A similar process takes place with external dependencies, such as when an application calls to an extension or another application.

MOD, MOD1, DLL and DLL1

Brew MP supports two module formats for the device, MOD and MOD1, and two DLL formats for the Simulator, DLL and DLL1.

Brew MP modules can by dynamically or statically linked with the platform.

- DLL for the Simulator and MOD for the device ensures backwards compatibility with BREW. Certain new Brew MP capabilities may not be available to MOD applications such as code-sharing or implementation for service classes.
- DLL1 for the Simulator, MOD1 for the device is an enhanced Brew MP format. All Brew MP capabilities are available to MOD1 applications.

The main difference between MOD and MOD1 is the file format. MOD is fully-linked executable code that can be run anywhere in RAM. MOD1 is in ELF format. The table below compares MOD and MOD1.

	MOD	MOD1
File format	Fully-linked executable code	ARM ELF
Module entry point	AEEMod_Load(IShell* piShell, void* pvtStdLib, IModule** ppiModule); 1. Provided by Generated code from IDE wizard 2. IShell object is the first object provided to every object when created. 3. Supports AEEStdLib.h	IMod_New(IEnv* piEnv, AEEIID id, void** ppiMod); 1. Provided by generated code from cifc.exe 2. IEnv object is the first object provided to every object when created. 3. AEEStdLib.h not supported with most of the alternatives available in AEEStd.h and AEEIEnv.h

	MOD	MOD1
Dependency on BREW application framework (BREW Shell)	Yes - classes in MOD are IShell-based and are only usable inside the BREW Shell.	No - classes in MOD1 are IEnv-based and are usable anywhere in the system.
Code sharing	Per-process copy of code in RAM	Single copy (if no writable data) in the entire system or per-Env copy (if writable data exists) of code in RAM
Code used for Service classes	No	Yes

The module loading functions are different. The MOD module loading function is inside AEEModGen.c when the IDE Wizard is used to generate the application framework. It shows that MOD receives a pointer to the BREW Shell and a pointer to AEEStdLib. Every class in MOD is given an IShell object when instantiated, and is an IShell-based class with a dependency on the BREW Shell.

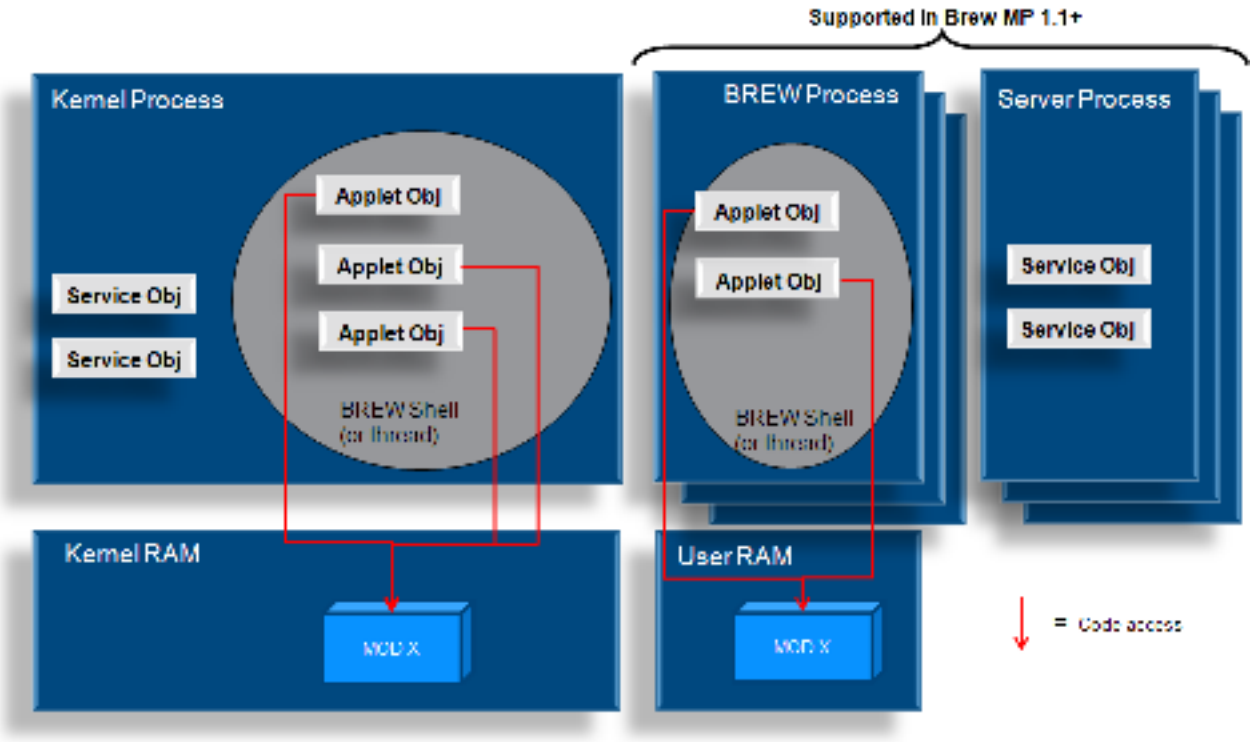
The MOD1 module loading function is auto-generated by the CIF compiler (cifc.exe). MOD1 receives a pointer to Env. Every class in MOD1 is given an IEnv object when instantiated, and is an IEnv-based class.

For MOD backward compatibility in BREW, the code is shared by objects within the same process. Brew MP copies the module to every process using the code from the module, as there is no way to determine whether there is any writable data in the MOD file.

Since MOD1 is ELF format, the header of the ELF reveals if there is any writable data in the module. This enables the system to determine whether to maintain only shareable copy in the entire system for module that contains no writable data, or to give every Env its own copy if the module is non-shareable due to writable data.

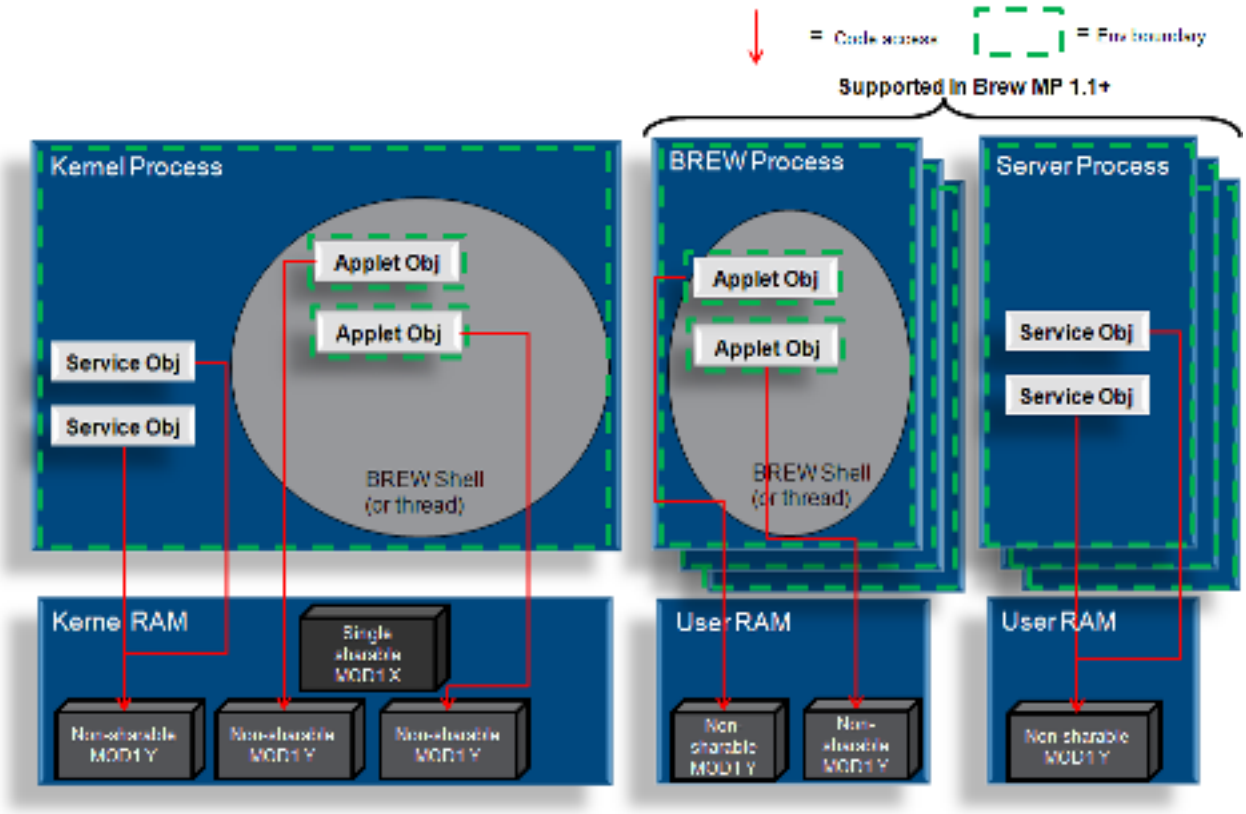
Mod contains only IShell-based classes, so it can only be used by objects inside the BREW Shell. Each process that uses MOD code receives its own copy. The following diagram shows code sharing for MOD.

Figure1. Code Sharing for MOD



MOD1 contains IEnv-based classes that can potentially be used anywhere in the system. If there is no writable data in MOD1, the system only maintains one shareable copy of the code in the entire system. If there is writable data in MOD1, it is considered non-shareable, and the system copies the file to each Env. The following diagram shows code sharing for MOD1.

Figure2. Code sharing for MOD1



For more information on MOD vs. MOD1, see the article *MOD vs. MOD1 in Brew MP* on the Brew MP Dev Net.

MIF and CIF

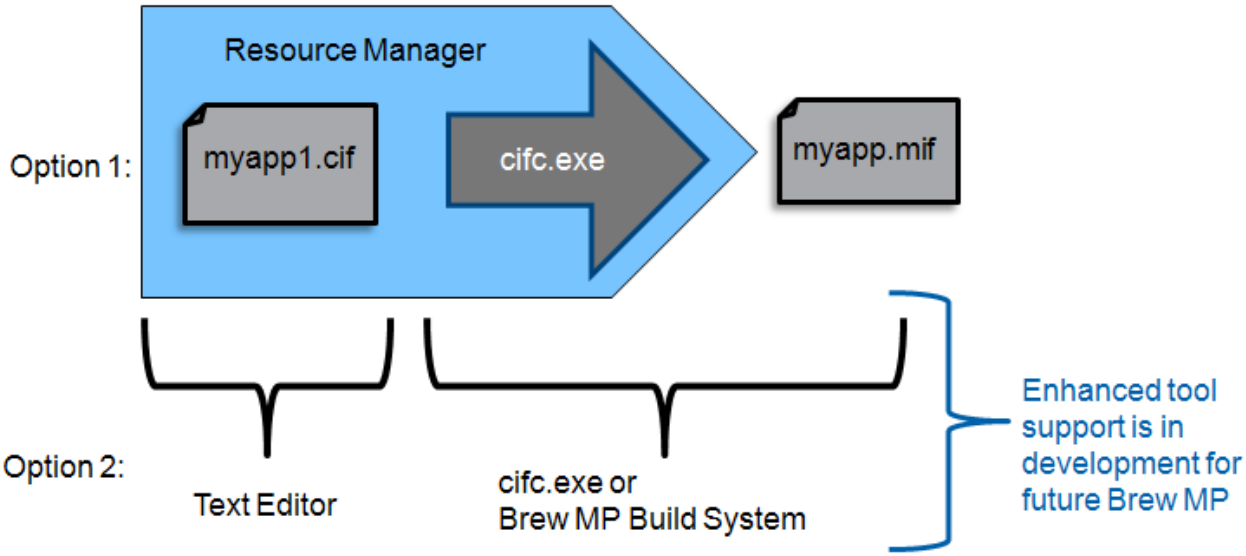
A Component Information File (CIF) is the source file for the Module Information File (MIF).

Component Information File (CIF)

A CIF specifies the privileges and resources for the module. CIFs are compiled into MIF using the CIF compiler, cifc.exe. CIFs are written in the LUA language, and are a replacement for the MFX file in BREW.

The Brew MP Resource Manager can be used to create CIFs. The Resource Manager provides a UI and a click-to-build interface. A second option to create CIFs is to use any text editor to manually create and manage the CIFs. To compile the CIF to MIF, use cifc.exe, or the Brew MP build system, which internally invokes cifc.exe.

Generally using Resource Manager to create CIFs is better suited to developers new to Brew MP. Once you are familiar with the CIF syntax, you can switch to using a text editor and cifc.exe or the Brew MP build system. CIF is basically a programming language, and it's difficult to wrap a UI around it, so the text-editing environment gives you more flexibility.



Module Information File (MIF)

A MIF contains module-specific information such as privileges, and resources such as applet names, in binary format. The MIF is the component used to identify external dependencies, in the form of external classes and extensions, that your application uses. The MIF is also used to provide classes and functionality to other applications.

The MIF can be used to specify the following:

- Icons for your application used on the device's Brew MP menu
- Copyright information
- External extensions (libraries) that your application uses
- Extensions (libraries) that your application exposes for others to use
- Application's unique ID (ClassID)

BAR and CAR

BAR and CAR files are for resource management.

Component Application Resource File (CAR)

CAR is the source file for BAR, compiled to BAR using cifc.exe. CAR is a replacement for the BRX file in BREW. A CAR contains ModRsc definitions for resources. A CAR should include a resource header file. The resource header file contains all the #define definitions for the IDS_XXXX used in CAR and C/C++ source code. If the CAR is managed by Resource Manager, the resource header is generated only to be used by C/C++ source code and is not included by the CAR.

A special case exists of a CIF, which contains only resource declarations.

Brew MP applications can be designed to execute on a variety of different devices, in a number of different languages. Resource files can be used to support the different devices and languages. By decoupling resources from your code and loading them at runtime, you can avoid cluttering your source code with a separate compilation flag for each supported language and device. To create a version of an application for a particular language or device, you need to create only system resource files instead of the entire application.

BREW Application Resource File (BAR)

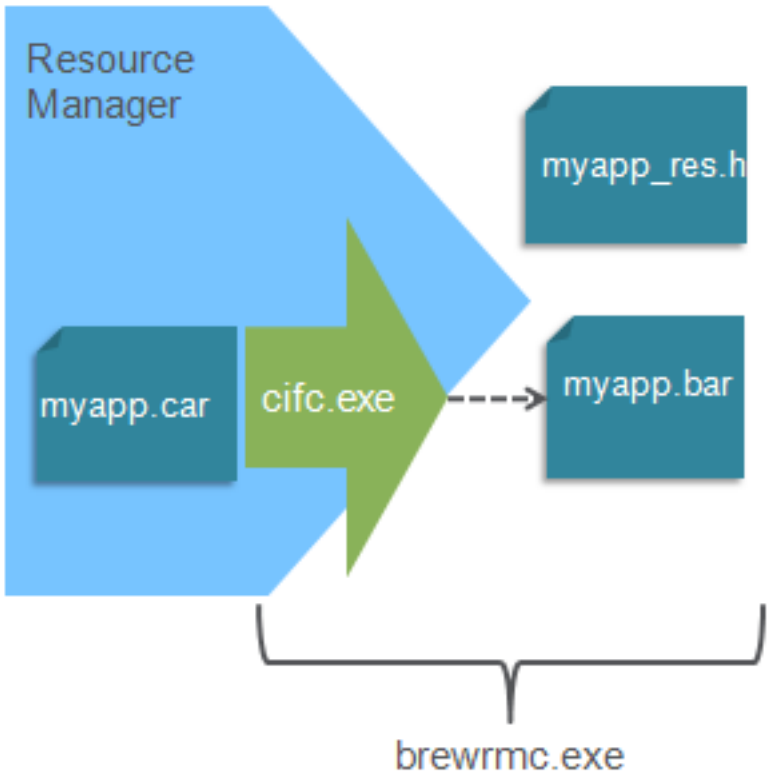
The BAR contains resources such as strings and images in binary format.

Creating resource files

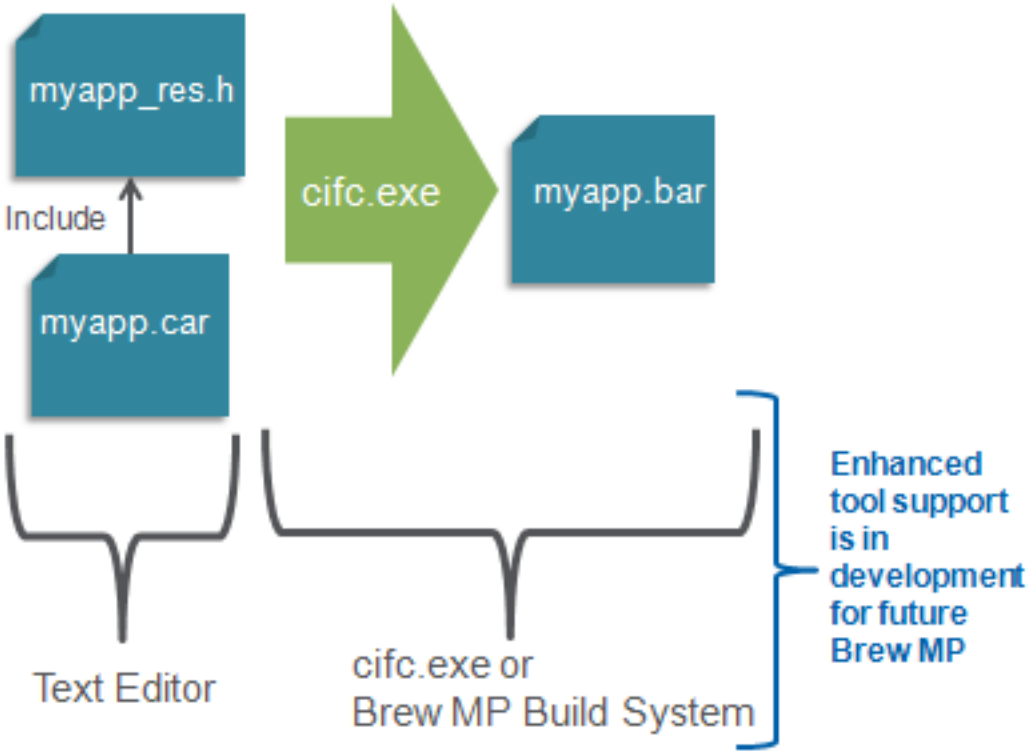
The following options are available to create and manage CAR/resource header files and compile them.

- Use Resource Manager, which provides a click-to-build UI. Brewrmc.exe is a command-line alternative to generate the resource header and compile the CAR to BAR. The resource header is not included in the CAR, it is auto-generated with the BAR.

  The Resource Manager can be used to create external resources used in applications such as strings, images, and binaries.



- Use any text editor to manually create and manage the CAR and resource header. To compile the CAR to BAR, use cifc.exe, or the Brew MP build system, which internally invokes cifc.exe, via Visual Studio or the Eclipse IDE, and command-line make.



For more information on CIF and CAR files, see the *Resource File and Markup Reference*, included with the Brew MP SDK and available on the Brew MP Dev Net.

Banned APIs

Certain functions should be used to avoid buffer overruns in code. The recommended replacements are available in AEEStdLib.h and AEEStd.h.

These functions are banned because a single call to them results in vulnerability. They are often not used in a safe way.

Banned function	comments	Replacements for MOD (in AEEStdLib.h)	Replacements for MOD1 (in AEEStd.h)
strcpy	Too easy to create a buffer overrun	STRLCPY	std_strlcpy
strcat	Too easy to create a buffer overrun	STRLCAT	std_strlcat
strncpy	Doesn't always NULL terminate	STRLCPY	std_strlcpy
strncat	Doesn't always NULL terminate	STRLCAT	std_strlcat
wstrcpy	Too easy to create a buffer overrun	WSTRLCPY	std_wstrlcpy

Banned function	comments	Replacements for MOD (in AEEStdLib.h)	Replacements for MOD1 (in AEEStd.h)
wstrcat	Too easy to create a buffer overrun	WSTRLCAT	std_wstrlcat
wstrncpy	Doesn't always NULL terminate	WSTRLCPY	std_wstrlcpy
wstrncat	Doesn't always NULL terminate	WSTRLCAT	std_wstrlcat
sprintf	Difficult to avoid buffer overruns with complex format strings	SNPRINTF	std_strlprintf
vsprintf	Difficult to avoid buffer overruns with complex format strings	VSNPRINTF	std_vstrlprintf
wsprintf	Difficult to avoid buffer overruns with complex format strings	WSPRINTF (it is different from wsprintf)	none
gets	Unsafe	gets was never in BREW	gets was never in CS
strtok	Not re-entrant	none	std_strchrsend (similar functionality)
scanf	Unsafe	scanf was never in BREW	std_scanul

Deprecated APIs

With the advent of Brew MP and lineage of prior BREW releases, certain APIs are deprecated. These deprecated APIs are superceded with newer APIs. The older APIs may still be used, though it is highly recommended that the newer APIs be used instead.

Deprecated APIs are documented in the *C/C++ API Reference*.

Families

Brew MP provides a variety of system-level functions and services to facilitate the development of applets for Brew MP-enabled devices. Brew MP modules can contain one or more applets or classes. These classes are exposed by a module at runtime and are loaded or unloaded on an as-needed basis.

Interface Services and Descriptions

Brew MP's AEE offers a number of distinct categories of services. The services provided, and the names of the interfaces that implement those services, are listed in the table below.

Family	Description	Interfaces
Connectivity	Bluetooth, WIFI and other methods of local area connectivity	IRemoteControl, IWIFI, wlan, IBTServiceDiscovery
Databases	Data storage on a mobile device, such as SQLite database support, call history, personal contacts, and timezone information	ITimeZone, IGallery, dbc_IConnect, pim_IContacts, ICallHIstory
Hardware	Managing the hardware of a mobile device, such as battery, camera, position determination, USB, FM radio, joysticks, and flip-phones	IBacklight, IBattery, ICamera, IFlip, IJoystick, IFMRadio, IHID, IKeysMapping, IPosDet, ISensorUtil, IUSBDevice
Languages	Adobe Flash, Trigs, Lua, and Java application management	IASArgs, IFlashPlayer, jams_IApp, ILua, ICachingResFile, IActorContext, ITrig
Multimedia	Support for multimedia content , including music, images and videos	IUnzipAStream, IContentMetaData, drm_IRightsChange, IImage, IMedia, ISound, IVocoder
Networking	DNS operations, multicast groups, the network subsystem on the mobile device, TCP and UDP sockets, and network connectivity.	IDNS, INetMgr, ISocket, ISSL, IAddrInfo, IMcastSession, IQoSList, INetwork, IWeb
Security	Encryption, certificates, and public/private key exchanges	IBN, ICipher1, IHashCtx, IPubKey
System	The Application Execution Environment (AEE), core services, module management, file system, memory management, locales, and settings management	IShell, IApplet, IMod, IAppletCtl, IResourceStatus, IDeviceNotifier, IFile, IHeap, IPort, ISignal, INotifier, IEnv, IControl, IBASE, IControl, ISettings
Telephony	Telephony functionality of a mobile device including SMS messaging and call handling	ISMS, ICallMgr, IPhoneCtl
UI	Displays and bitmaps, graphics, fonts, UI Widgets, and window management	IDisplay, IGraphics, IDisplayCanvas, IFont, IBitmap, IResFile, IWidget,

For more information

Brew MP documentation

Brew MP documentation is available on the Brew MP Developer Network.

There are four categories of Brew MP documentation:

- Primers guide you through installing the tools, setting up the work environment, writing a basic "Hello, World" application, and debugging it in the Simulator.
- Technology Guides help you understand Brew MP technologies and functional areas.
- How Tos provide solutions to specific programming problems, and include code snippets with explanations and sample code files that you can download.
- References provide the detailed information you need when working with the tools and writing Brew MP code to produce a successful application.

Brew MP sample code

The Brew MP SDK includes "hello world" type sample applications that you can study and use as a basis for your own applications. These sample applications correspond to primers and demonstrate Brew MP tools. The folder containing the sample code can be installed on your machine from the Brew MP SDK Manager, in the setup tab. Additional applications that demonstrate API usage, and leverage various Brew MP platform capabilities, are available on the Brew MP Dev Net.

In general, to view and edit the source code:

1. Run Visual Studio and open one of the sample project workspaces (*.sln and *vvproj). Sample .dsw and .dsp files for use with Microsoft Visual Studio have also been shipped with the SDK. Run the application in the Simulator.
2. Make a small change to the source code (for example, showing a different text message) and rebuild the application using the Build menu in your compiler. Be sure to backup any projects you use prior to making changes.
3. Run the Simulator, to verify that your change has taken effect.

While any development tool that can compile a Win32 DLL works, keep in mind the useful functionality provided by the Brew MP Visual Studio and Eclipse Plugins.

When looking at the code, you may find many familiar things, such as function calls, loops, switch statements. You may notice that the code is written in C with some C++ nuances.

Frequently asked questions

What are the differences between an extension and a service?

Extensions and services are two orthogonal concepts. One corresponds to the type of software module and the other corresponds to the type of class the software module contains. See the table below. An extension is a software module that does not implement the applet class, so it can only be loaded and executed by Brew MP when it is called by an application. It is similar in concept to software plugins in the PC world. A service is a type of Brew MP class. See the Classes on page 7 section for more details.

Brew MP Class types	Supporting Module Formats	Applicable Brew MP Software Module	Supporting IDE Wizards

Applet Class	MOD or MOD1	Application	Application (Applet Class)
In-Process Class	MOD or MOD1	Application or Extension	Extension (In-Process Class)
Service Class	MOD1	Application or Extension	Extension (Service Class)

- Brew MP class types: To write C/C++ programs in Brew MP is to implement three types of Brew MP Classes: applet class, in-process class, and service class
- Supporting module formats: There are two types of module formats supported by Brew MP that contain one or more Brew MP classes, MOD and MOD1. MOD can only contain applet and in-process classes and MOD1 can contain all three types of classes. For more information on MOD and MOD1, see the *MOD vs. MOD1 in Brew MP Technology Guide*.
- Applicable Brew MP software modules: A Brew MP application is a software module that contains at least one applet class so that it can be loaded and executed directly by Brew MP. A Brew MP extension is a software module that contains only non-applet classes and therefore can only be loaded and executed by Brew MP when it is called by a Brew MP application.
- Supporting IDE Wizards: Wizards are provided by the Brew MP IDE plugins to assist developers in auto-generating code for Brew MP classes in specified module formats. Currently, there are two wizards available for C Applications and C Extensions. The C Application wizard generates code for an applet class in MOD or MOD1 format. The C Extension wizard generates code for an In-process class in MOD or MOD1 format.

What's the relationship between and an applet and an application?

An applet is a Brew MP class that supports IApplet. It can be loaded and executed by BREW Shell and receive BREW EVT_XX events. An application is a software module that contains at least one applet class that can be loaded and executed by Brew MP. An application may also contain any number of in-process or service classes. Please see the Classes on page 7 section for more details.

What are the differences between an in-process class and a service class?

An in-process class is a class that can only be instantiated in the process (or more specifically, the same Env) of the caller. A service class is a class that is instantiated in a designated process (specified in the CIF in which the service class is declared). A service class must support one or more interfaces that are remotable because the caller may call from a different process than the service object. For more information, see the Classes on page 7 section of this document, and the *Resource Files and Markup Reference* document on the Brew MP Developer Network.

Why should I use a service class instead of an in-process class?

- When you would like your class to be accessible to user mode.
- To be instantiated and protected in a designated process rather than in the caller's environment, as is the case for an in-process class.
- To be preemptively multi-threaded.
- If you manage shared data between clients and manage access to shared resources.

How does an applet interact with a service or an in-process class?

An applet calls IShell_CreateInstance() or IEnv_CreateInstance() to instantiate an in-process or a service class. When an applet interacts with an in-process class, all the function calls are direct invocations because they are in-process. When an applet interacts with service classes, the calls are remote

invocations because service objects are not in the same context or environment of the calling applet. See the Classes on page 7 section for more information.

How does an applet interact with another applet?

Applets can only "start" another applet by calling IShell_StartApplet() or related APIs, but not by calling IShell_CreateInstance() or IEnv_CreateInstance(). The applets interact with one another by posting events (for example, using IShell_PostEvents()), shared files (permitted in FS_ACL_Grant in CIF), IFIFO IPC communication (permitted in FIFO_ACL_Grant in CIF), or a singleton service object, etc.

How are resources tracked and managed by applet, in-process, and services classes?

Resources that are loaded by an in-process class are always loaded in the context, and share the privileges of, the invoking application. Memory allocated as a result of resource loading is always tracked in the context of that application. In-process objects loaded by one application or extension in one process should not be sent to another application in a separate process (e.g. IShell_SendEvent()) since it is unsafe and violates the memory protection boundary. Applet and service objects have their own sets of privileges and are hosted in their designated processes for their own resource tracking. Remotable objects can be passed between service objects or between a service and an applet across process boundaries.

Why should I create a Brew MP extension instead of an application?

An application can be loaded and executed directly by Brew MP like any executable in the PC world. An extension is similar to a code extension or software plugin. Extensions can be re-used by various applications so the code doesn't need to be duplicated or re-implemented in the module of each using application.

Can I package an applet class, an in-process class, and a service class into a single module?

Yes. Extensions and applications are essentially software modules or repositories of classes. An application has at least one applet class and any number of in-process and/or service classes. An extension can have any number of in-process and/or service classes but cannot contain applet classes. Currently, if the application or extension is to be distributed through QIS BDS, applet classes and non-applet classes must be in separate modules. This is a BDS restriction rather than a Brew MP restriction.