A company of SIM Tech

SIM5360 BMP Demo Architecture&Programming Model Note V1.00

Document Title:	SIM5360 BMP Demo Architecture&Programming Model Note
Version:	1.00
Date:	2012-09-28
Status:	Release
Document Control ID:	SIM5360_BMP_Demo_Architecture&Programming_Model_Note_V1.00

General Notes

SIMCom offers this information as a service to its customers, to support application and engineering efforts that use the products designed by SIMCom. The information provided is based upon requirements specifically provided to SIMCom by the customers. SIMCom has not undertaken any independent search for additional relevant information, including any information that may be in the customer's possession. Furthermore, system validation of this product designed by SIMCom within a larger electronic system remains the responsibility of the customer or the customer's system integrator. All specifications supplied herein are subject to change.

Copyright

CONTENTS

Version history

Date	Version	Description of change	Author
2013-12-09	1.00	Origin	qiujianhua

SCOPE

This document describes how to install Brew MP Application developer environment and how to use SDK build a application. This document is subject to change without notice at any time.

1. INTRODUCTION

Binary Runtime Environment for Wireless Mobile Platform (Brew MP) is an application development platform created by Qualcomm。It's a freely available mobile Operating System(OS) platform.

2. BREW MP ARCHITECTURE

Brew MP have 4 core Softeware Layers(Figure 1)

- OS Services: Abstract the kernel and memory management ,Provide component management, process, and security for the platform. Provide portability for chip set.
- Platform Services: Contain modem, multimedia, and general service features, it's also the layer of the Brew MP API.
- Application Environment: Foundation for applications running on Brew MP,Support Flash,Lua,TrigML™ , Widgets(BUIW) and Windows Manager
- Application: Static Apps and 3rd Party Apps. It could be developed by C/C++, Flash,TrigML or Java.

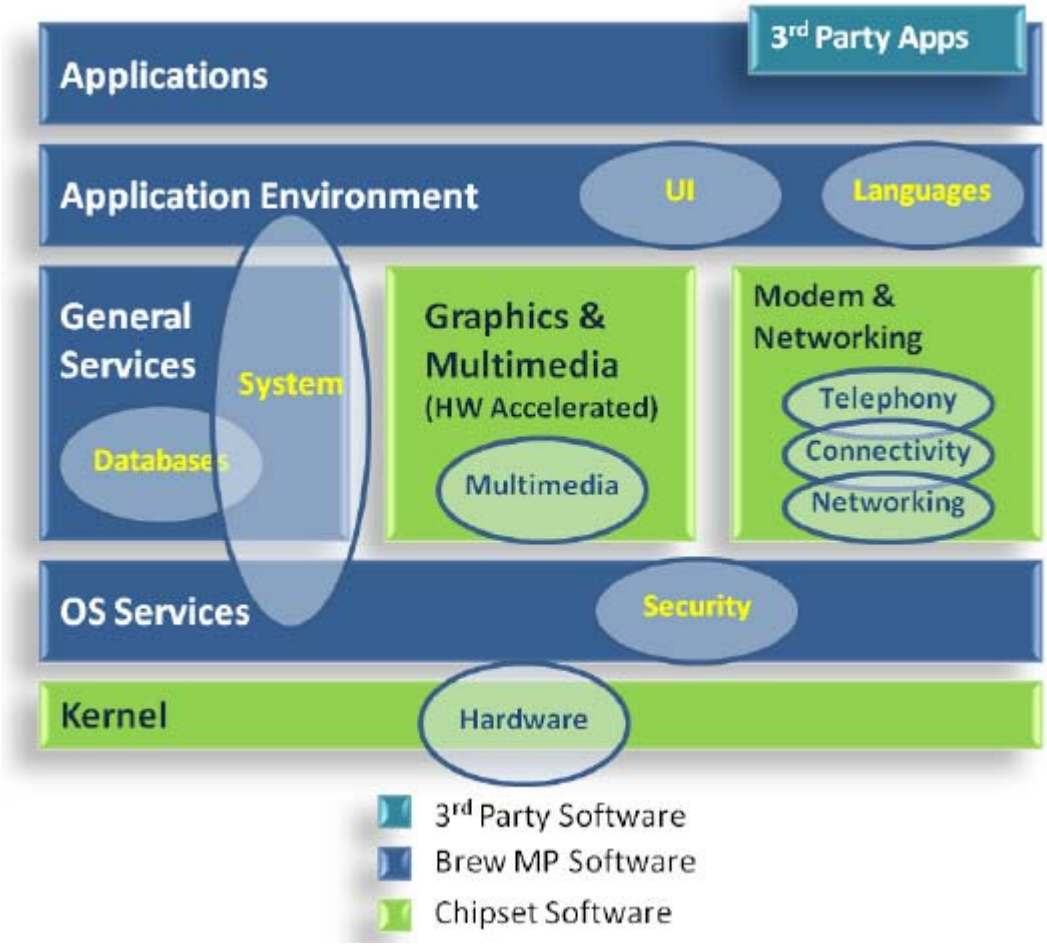


Figure 1

Brew MP application model is event driven. Application response from the operating system events. Brew MP application does not contain a main program loop, All input is controlled by event. This model requires only minimal system resources can realize the efficient and smooth execution, and can realize the simple development based on task.

Brew MP use the object-oriented modular design, controllable system structure, security and services. It provide cross language environment (C/C++, Flash) dynamic platform.

3. PROGRAMMING MODEL

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:

3.1 Interface

An interface is a software contract between an implementing class and the client that uses the interface. Interfaces provide the definition of a particular set 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 https://brewx.qualcomm.com/classid/.
The following is an example of an 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 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.

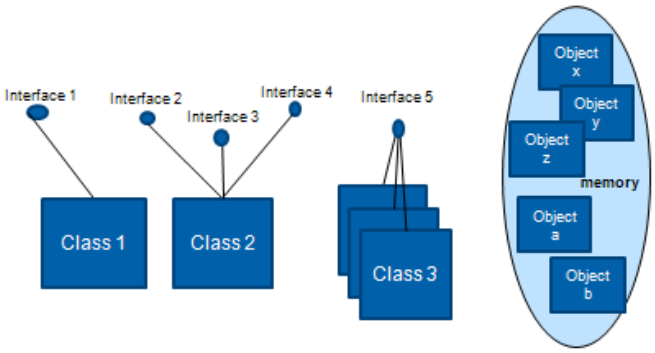


Figure 2

Defining interface 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. For more information on IDL, see the *QIDL Reference*, and the QIDL Compiler section of the Brew

MP *Tools Reference*, both of which are inhttp://developer.brewmp.com/resources on the Brew MP website.

Reference counting:

Brew MP objects use reference counting and are released when the count is zero.

An object's reference count is updated at the following times:

- When the object is instantiated by calling ISHELL_CreateInstance() or IEnv_CreateInstance(), the reference count of the object is incremented.
- When a pointer to the object is retrieved using one of the xxxxx_QueryInterface() function calls, or other calls that return object pointers, the reference count of the object is incremented.
- When the reference count is explicitly incremented or decremented using xxxxx_AddRef() or
- xxxxx_Release() function calls.

Most interfaces provide AddRef() and Release() functions, which can be used to increment or decrement the reference count, respectively. Applications should call the AddRef() function when they store a pointer to an object in the application's data structure and call the Release() function when the pointer is no longer needed.

Applications should also release all reference counted items in their FreeAppData() function. For example, if an application instantiates AEECLSID_SysClock and AEECLSID_ALARMMGR, as follows:

```
if (ISHELL_CreateInstance(pMe->piShell, AEECLSID_SysClock,
                                (void **)&pMe->piUserClock) != SUCCESS) {
    pMe->piUserClock = NULL;
    return FALSE;
}
if (ISHELL_CreateInstance(pMe->piShell, AEECLSID_ALARMMGR,
                                (void **)&pMe->piAlarmMgr) != SUCCESS) {
    pMe->piAlarmMgr = NULL;
    return FALSE;
}
```

The application's FreeAppData() function needs to free the resources for these objects, as follows:

```
void c_sample_FreeAppData(c_time_examples * pMe)
{
    if (pMe->piUserClock != NULL) {
        ISysClock_Release(pMe->piUserClock);
        pMe->piUserClock = NULL;
    }
    if (pMe->piAlarmMgr != NULL) {
        IALARMMGR_Release(pMe->piAlarmMgr);
        pMe->piAlarmMgr = NULL;
    }
}
```

3

In Brew MP, software programs are written as classes. A class is a user-defined type that encapsulates data and behavior (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 CIF/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. The vtable contains the addresses of the functions provided by the class.
C/C++ programs in Brew MP implement three types of Brew MP classes:

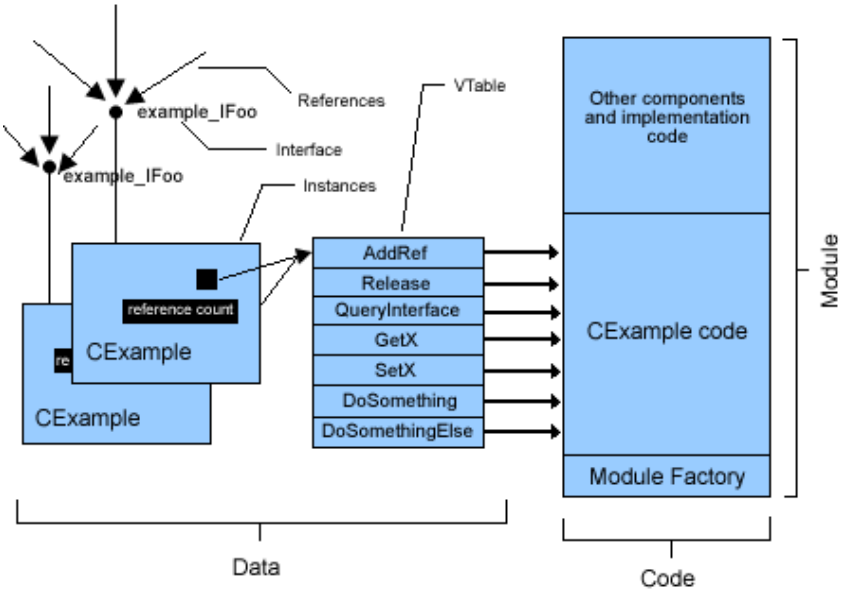
- Applet classes
- In-process classes
- Service classes

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.

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

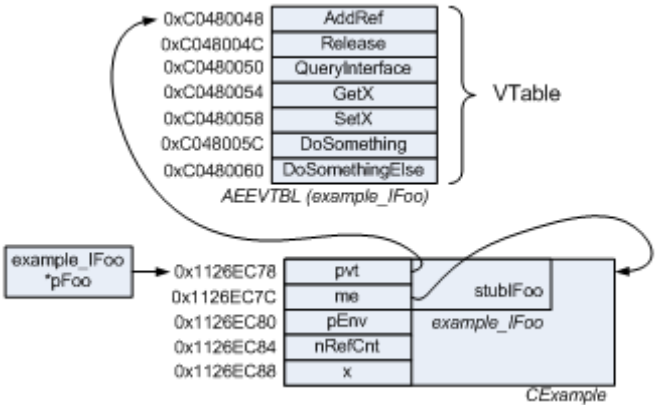
Brew MP class types	Supporting module formats	Applicable Brew MP module types	Supporting IDE Wizards
Applet class	MOD or MOD1	Application	Application (applet class)
In-Process class	MOD or MOD1	Application or extension	Extension (in-process class)
Service Class	MOD1	Application or extension	through Add->Class->Service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

In the figure below, the interface name is example_IFoo, and the instantiated class (object) is 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 figure below, example_IFoo creates an instance of CExample and locally stores pvt (the pointer to the Vtable). The second field stores me, which is a pointer to the actual instance of CExample.



Typically, a class can contain one or more interfaces. See Interfaces on page 15 for more information. There are two commonly used mechanisms for creating instances of a class with QCM interfaces.

1. Using the IShell_CreateInstance() or IEnv_CreateInstance(). Classes with QCM interfaces may register (or advertise) in the CIF/MIF. 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.

Applet class:

Applet classes implement IApplet and are identified, dynamically discovered, and instantiated using a unique Applet ID. 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.

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.

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-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. The object 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.
The following is an example of declaring an in-process class in CIF for a MOD1:

```
Class {
      classid = AEECLSID_MyClass,
      Programming Model for Developers Brew MP programming concepts and terminology
      Qualcomm Confidential and Proprietary | © 2012 QUALCOMM Incorporated 20
      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.

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.

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 only execution context that is supported for a service class is the Kernel process, which is statically specified in the CIF of the containing module.

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.

Service classes that provide QCM interfaces can be registered in the CIF/MIF using the Service primitive.

The service object is created in the designated 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. 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). Each service class can also specify the privileges the caller must possess to access the service 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.

- **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 preemptive 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.

- **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

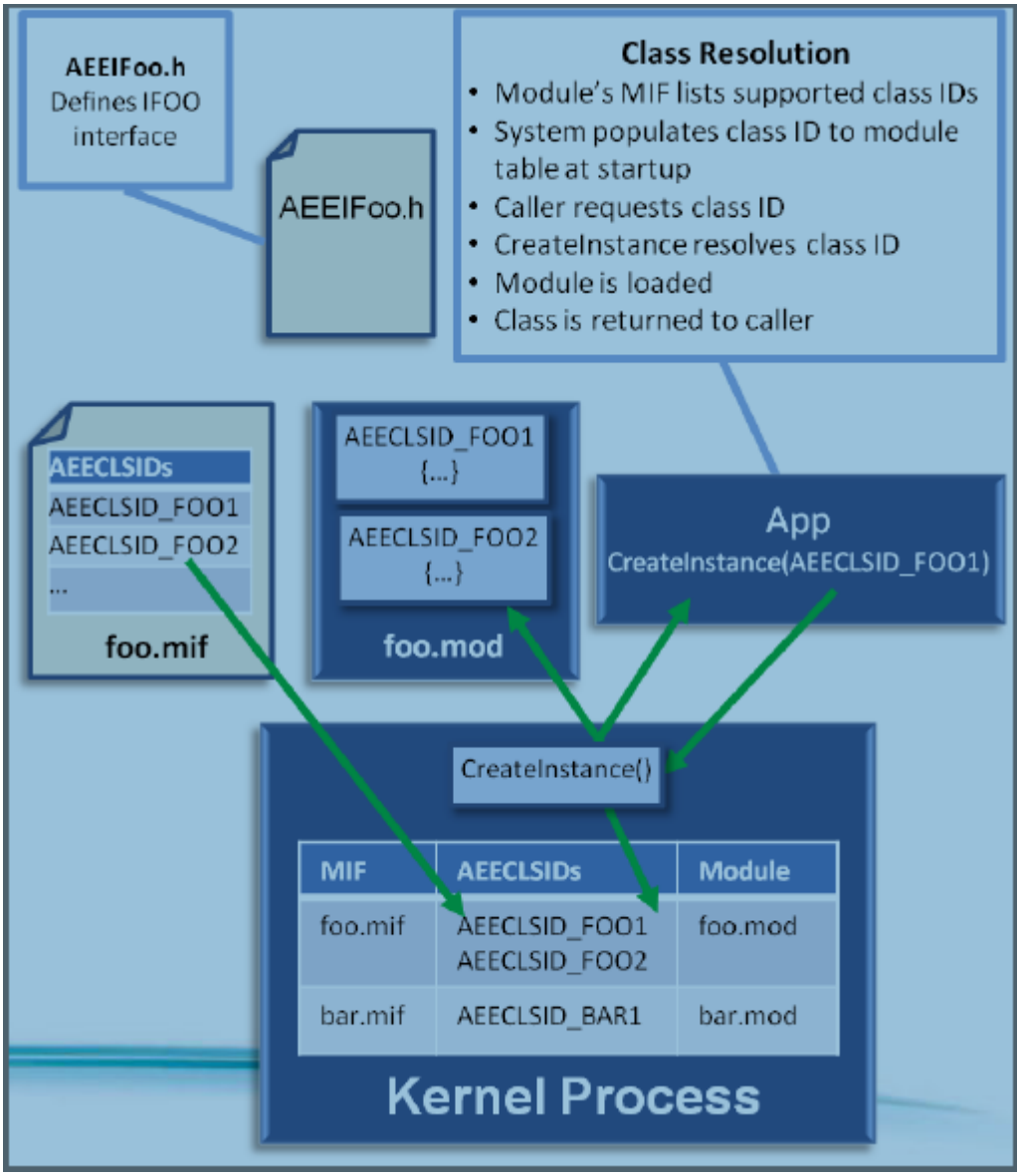
the inprocess class definition.

```
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.
- AEEIID_MyService specifies the default interface (defined in IDL and remotable) that the service class implements.

Class resolution in Brew MP:

When an application calls IShell_CreateInstance() or IEnv_CreateInstance() in instantiate an instance of a class, the specified ClassID must be resolved to the appropriate class. 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.

3.3 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 do not have 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 a software distribution format, which includes the MOD file (the executable), the MIF, and any other optional resource files, such as a signature file. The MOD file is 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.

A module can contain other files in its module directory. These files can be part of the initial (install-time) state of the module or may be created after the module is installed. Modules can be statically linked (LIB files), or stored in the file system (EFS) as dynamically loaded modules.

- All dynamic modules are digitally signed. A SIG file provides a digital signature that verifies that the module is from a trusted source.
- 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 (such as 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.

Static modules:

Static modules are linked into the boot (or flash) image so that OS Services does not have to load the module binary into memory at run-time. The boot image can be defined as the software image into which the OS Services libraries themselves are statically linked. Static modules are compiled and linked into the firmware image in the same manner as device software and OS Services itself, and they live along side OS Services. The default static modules are the file system, signature code, and core services.

The module parts of a static module are statically linked, the MIF is stored as byte array, and there is no signature file. To make statically linked code available to the OS Services environment, a

statically linked IModule implementation is provided that links directly with the static classes. Static classes are listed in an array of structures as in earlier versions of BREW. As with dynamic code, all objects of a class share one IModule instance per process, and that instance can be used to share data that is instantiated once per process.

Dynamic modules:

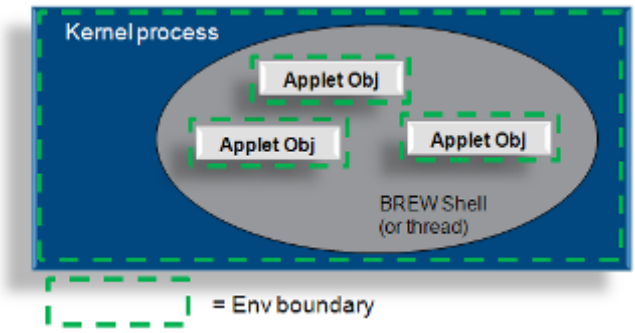
A dynamic module is a group of files that must contain a Module Information File (MIF), and can contain any of the following:

- A module file
- A signature file
- Zero or more resource files
- Any other files specific to the module's execution

Dynamic modules are stored on the device as files in the file system, and read into memory when needed by OS Services.

4. RUNTIME ENVIRONMENT

In a deployed Brew MP device, the runtime configuration is expected to consist of a single kernel process hosting the component infrastructure and privileged services. In Brew MP, applications are implemented as applet objects, which are loaded and executed in their own environments under the Brew application framework (Brew Shell), as shown in the figure below.



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 . 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 is hosted in the kernel process, which 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 that has the privileges to

use this service. A server resource describes the parameters that are passed to create the server, including, most importantly, the set of privileges that are granted to the server process. A service Resource 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.

5. 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.

There is one Env per applet and per process (per kernel process). Brew MP supports multiple applets running in the same process, so each 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 context for an applet. The memory protection domain is established by the Env of the process in which the applet resides, and therefore applets in the same process still share the same heap memory.

6. 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 applets 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.

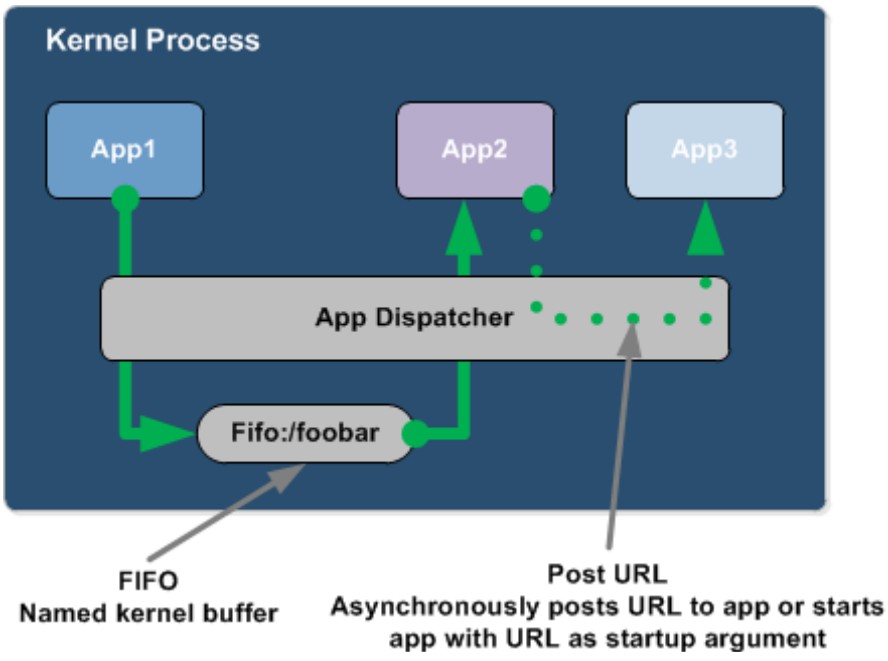
The Brew MP system process model transparently supports memory-protected (multi-process), non memory-protected (single-process), single processor, and dual processor implementations. In modern operating systems, there is usually one kernel process and multiple user processes. For both performance and resources considerations, Brew MP only supports the kernel process and no user processes are supported. .

7. 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.

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(). ISHELL_PostURL() queues an event which later causes the application to be loaded. The application can then choose to start if desired. ISHELL_BrowseURL() synchronously loads and starts the associated application with the URL. Brew MP supports a number of pre-defined URLs.



8. 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.

8.1 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

Multitouch events are also generated by touch-enabled devices:

- EVT_POINTER_MT_DOWN
- EVT_POINTER_MT_MOVE
- EVT_POINTER_MT_UP

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

8.2 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 key press 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

8.3 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.

The sequence of a key press 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.

8.4 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.

8.5 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.

9. HOW APPLICATIONS ARE SUSPENDED AND RESUMED

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 application should do the following:

- 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.

Contact us:

Shanghai SIMCom Wireless Solutions Ltd.

Add: Building A, SIM Technology Building, No.633 Jinzhong Road, Changning District, Shanghai, P. R. China 200335

Tel: +86 21 3252 3300

Fax: +86 21 3252 3020

URL: www.sim.com/wm