



# **ZOOM OUT BREW 2008 CONFERENCE**

**Developing Components –  
Techniques, Tools and Testing**

Jack Brennen, Sr. Staff Engineer  
Qualcomm Incorporated

**QUALCOMM**



## What Will I Learn Today?

- Review BREW<sup>®</sup> Mobile Platform Component Model
- How to use Qualcomm-supplied tools
  - QIDL: Qualcomm Interface Definition Language
  - CIFC: Component Information File Compiler
  - CTF: Component Test Framework
- Walk through the implementation of a new component (and a new interface) including all concepts above



## BREW Mobile Platform Component Model

- Object Oriented
- Similar to COM or CORBA
- Objects are Polymorphic; all object methods are resolved at run-time
- Each Object supports one or more Interfaces
- Interfaces are similar to C++ abstract base classes; an interface isn't a declarable object on its own, but rather a property of an Object



## BREW Mobile Platform Component Model

- Remote Objects are supported
- An Object and its client need not be aware of each other's "location" – they may be running on different processors entirely
- In order for an Object to be used remotely, its interfaces must be "remotable" – we'll talk about this later



## BREW Mobile Platform Component Model

- In BREW, a “Class” is a mapping between a unique “Class ID” and a particular method for instantiating a particular type of object
- For instance, one built-in Class is the SysTimer class; the Class ID is AEECLSID\_SysTimer and it maps to a method for instantiating an object which supports the ISysTimer interface



## BREW Mobile Platform Component Model

- Interfaces have IDs as well; every Object supports a QueryInterface method
- QueryInterface is used to “ask” an Object whether it supports a particular interface
- QueryInterface is most commonly used when an Object supports multiple interfaces, but its existence is mandatory for all Objects



## BREW Mobile Platform Component Model

- Class IDs and Interface IDs are required to be unique within the system
- To help ensure unique IDs globally, Qualcomm provides an ID allocation service on its Developer web site
- Registered developers can allocate their own IDs which are guaranteed to be unique



## Let's Design a New Class

- The existing interface ISignal is a built-in interface used for communication
- ISignal only supports one class-specific method: ISignal\_Set(), which takes no arguments
- When ISignal\_Set() is called, generally some object somewhere receives a notification, but that's outside the scope of the ISignal interface definition and is not important to us yet



## Let's Design a New Class

- Our new object will implement a “stack” of ISignal interfaces
- Three methods required:
  - Set(): invoke ISignal\_Set() on the topmost ISignal
  - Push(): push an ISignal interface onto the stack
  - Pop(): pop the topmost ISignal interface from the stack
- Call the new interface ISignalStack
- Derive the new interface from ISignal, adding the Push() and Pop() methods



## Let's Design a New Class

- Our first step is to write an IDL (Interface Definition Language) description of ISignalStack
- ISignalStack is based off of ISignal, with two extra methods (Push and Pop) and it requires a new unique interface ID
- An interface ID was retrieved from the BREW Developer's web site



## ISignalStack.idl

```
#ifndef AEEISIGNALSTACK_IDL
#define AEEISIGNALSTACK_IDL
#include "AEEISignal.idl" // inherit from ISignal

const AEEIID AEEIID_ISignalStack = 0x01072928;

interface ISignalStack : ISignal
{
    AEEResult Push(in notnil ISignal signal);
    AEEResult Pop();
};
#endif /* AEEISIGNALSTACK_IDL */
```

## What does the QIDL compiler do?

- The QIDL compiler will generate three different outputs from our IDL file:
  - ISignalStack.h: standard C include file needed for C source that wants to use ISignalStack
  - ISignalStack\_{stub,skel}.{c,h}: source code; code in these files is used when invoking a remote ISignalStack object
  - ISignalStack.cif: a Component Information File describing the remoted ISignalStack interface



## Why use QIDL?

- Writing the C include file, the remoting code, and the remoting CIF file is usually not much fun when you have to do it by hand
- The QIDL compiler will validate that your interface is remotable
- Remotable? We talked about that before, right?  
Let's talk about remotable interfaces...



## Remotable Interfaces

- QIDL only allows remotable interfaces
- Remotable interface methods: two main restrictions
  - Return value must be of type AEEResult; this is because remotable methods may return errors related to the remoting mechanism, which are of type AEEResult
  - In general, pointers may not be passed in or out; the object and its client may not be in the same address space
  - One type of pointer is allowed: a pointer to a remotable interface; the remoting mechanism will convert this to an equivalent interface pointer at the remote end



## How Does Remoting Actually Work?

- Remotable interfaces require substantial support from the infrastructure
- Most of the support happens “behind the scenes”
- It’s possible to implement and use remotable interfaces without knowing much about the details
  - Follow the rules for remotable interfaces
  - Use the Qualcomm-provided QIDL compiler



## Remotable Interfaces

- Allowable arguments to a remotable interface:
  - Scalars (integers, floats, bytes, chars, and wide chars)
  - Remotable interface pointers (optionally allowing NULL)
  - Structs or Unions of the above
  - Arrays or Sequences of the above (a Sequence is a variable length array)
  - Strings or Wide Strings (zero-terminated Sequences)
  - Nested aggregations (arrays or sequences of structs, structs may contain arrays, etc.)



## Component Information File

- Now that we've finished the interface design, we need to describe the class using a Component Information File (CIF)
- The CIF describes the supported class IDs and for each one, how to create an instance of that class
- Classes may be created locally (in the creator's process), in the kernel, or in a separate server process



## SignalStack.cif (Part 1)

```
include "AEEServerHost.bid" -- ServerHost Class ID
include "SignalStack.bid"   -- SignalStack Class IDs
include "SignalStack.h"    -- Prototype for SignalStack_New
include "ISignalStack.h"   -- QIDL-generated header
```

```
Class {
    classid = AEECLSID_SignalStack,
    newfunc = SignalStack_New
}
```



## SignalStack.cif (Part 2)

```
Service {  
    serviceid = AEECLSID_SignalStack_K,  
    iid = AEEIID_ISignalStack,  
    serverid = 0,  
    servedclassid = AEECLSID_SignalStack,  
    required_privs = {0}  
}
```

```
Service {  
    serviceid = AEECLSID_SignalStack_U,  
    iid = AEEIID_ISignalStack,  
    serverid = AEECLSID_SignalStack_Server,  
    servedclassid = AEECLSID_SignalStack,  
    required_privs = {0}  
}
```



## SignalStack.cif (Part 3)

```
Server {  
    serverid = AEECLSID_SignalStack_Server,  
    mainid = AEECLSID_ServerHost,  
    priority = 81,  
    stacksize = 8192,  
    maxservices = 512,  
    maxmemory = 0x20000,  
    args = "threads=1;stacksize=8192;priority=50";  
    privs = { AEECLSID_SignalStack_Server }  
}
```



## We need Class IDs

- In our CIF file, we referenced four class IDs related to SignalStack
- Go to the BREW Developer's web site and get four class IDs
- Create a BID file with the class IDs – this is just a C include file with #define directives



## SignalStack.bid

```
#define AEECLSID_SignalStack      0x01072929
```

```
#define AEECLSID_SignalStack_K    0x0107292A
```

```
#define AEECLSID_SignalStack_U    0x0107292B
```

```
#define AEECLSID_SignalStack_Server 0x01072949
```



## SignalStack.h

```
#include "AEEStdDef.h"
```

```
#include "AEEIEnv.h"
```

```
// Only one public function needs to
```

```
// be exported for the SignalStack class
```

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





## Makefile (part 2)

```
BUILD_MODS = SignalStack
```

```
SignalStack_LIBS = a1mod a1std
```

```
SignalStack_IMPLIBS = a1_imp
```

```
SignalStack_IDLS = ISignalStack
```

```
SignalStack_OBJS = SignalStack
```

```
SignalStack_CIFS = SignalStack
```

```
INCDIRS=$(APIONE_DIR)/inc
```

```
include $(RULES_MIN)
```



## Ready to write C implementation

- Infrastructure is done
  - IDL file describes the interface
  - CIF file describes the classes and how to create them
  - BID file contains the class IDs
  - H file contains the prototype for the class “New()” function
  - The Makefile takes care of the mechanics of building



## Component Test Framework

- Now that we're done coding our class, how do we test it?
- CTF is the standard test framework
- CTF is implemented within the QCM Component Model of course



## Component Test Framework

- To use CTF, you write a test class which is a standard QCM class
- Your test class must have a class ID, and must implement the ICTSuite interface
- Start a program running the Qualcomm-supplied CTFHost class; pass it the ID of your test class
- The CTFHost class drives the test procedures



## ICTSuite

- ICTSuite is implemented by all CTF test suites
- ICTSuite is not a remotable interface
- Three class-specific methods
  - GetModuleName() returns an ASCII string
  - InitTests() is invoked to indicate that a test suite is about to be executed
  - End() is invoked after tests are complete to allow the test suite to perform any necessary cleanup



## ICTF

- When the ICTSuite\_InitTests() method is invoked, an ICTF interface is provided to the test suite
- ICTF is not a remotable interface
- This ICTF interface pointer must be stored; it will not be provided along with each individual test
- The ICTF interface allows the test suite to communicate back to the CTFHost program



## ICTF Methods

- AddTests(): add a set of tests to the test suite
- AddTest(): add a single test to the test suite
- Log(): output an ASCII string in the test log
- LogV(): output a printf-style formatted message in the test log
- EndTest(): indicate that the current test has completed



## ICTF Tests

- To register a test with CTF, you give an ASCII string describing the test and a function pointer of the test entry point
- The test function has one argument – the ICTSuite interface pointer of your own class
- Tests do not terminate when your test function returns; they only terminate when you invoke `ICTF_EndTest()`



## Component Test Framework

- You need most of the same infrastructure files for your test component that we used for making the class component
- No IDL file is needed; your test component does not introduce a new interface
- The CIF file, BID file, H file, and Makefile are similar to the ones we did for the class component



## SignalStackTest.cif (Part 1)

```
include "SignalStackTest.bid"
```

```
include "SignalStackTest.h"
```

```
include "CTFHost.bid"
```

```
Class {
```

```
    classid = AEECLSID_SignalStackTest,
```

```
    newfunc = SignalStackTest_New
```

```
}
```



## SignalStackTest.cif (Part 2)

```
Program {  
    programname = "SignalStackTest",  
    mainid = AEECLSID_CTFHOST,  
    priority = 13,  
    stacksize = 8192,  
    maxservices = 512,  
    maxmemory = 0x200000,  
    args = string.format("SignalStackTest 0x%x",  
        AEECLSID_SignalStackTest),  
    privs = { AEECLSID_CTFHOST, AEECLSID_SignalStackTest }  
}
```



## SignalStackTest.bid

```
#define AEECLSID_SignalStackTest 0x0107292C
```



## SignalStackTest.h

```
#include "AEEStdDef.h"
```

```
#include "AEEIEnv.h"
```

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





## Makefile (part 2)

```
BUILD_MODS = SignalStackTest
```

```
SignalStackTest_LIBS = a1mod
```

```
SignalStackTest_OBJS = SignalStackTest
```

```
SignalStackTest_CIFS = SignalStackTest
```

```
INCDIRS=$(APIONE_DIR)/inc $(CTF_DIR)/inc
```

```
include $(RULES_MIN)
```



## Ready to write C code for CTF test

- Infrastructure is done
  - CIF file describes the test class and how to start CTF
  - BID file contains the test class ID
  - H file contains the prototype for the class “New()” function
  - The Makefile takes care of the mechanics of building

# Output from CTF execution

```
CTFHost.c:173 (High) 21:11:25:609 - ProgramName: SignalStackTest
CTFHost.c:173 (High) 21:11:25:609 - Suite: SignalStackTest
CTFHost.c:173 (High) 21:11:25:609 - Test: SignalStackTest_Basic
CTFHost.c:173 (High) 21:11:25:718 - Signal callback - signal delivered
CTFHost.c:173 (High) 21:11:25:718 - SignalStackTest_Basic passed
CTFHost.c:173 (High) 21:11:25:718 - Signal was delivered 1 times
CTFHost.c:173 (High) 21:11:25:718 - Result: Pass
CTFHost.c:173 (High) 21:11:25:718 - [VERBOSE]
CTFHost.c:173 (High) 21:11:25:718 - SignalStackTest Result Summary:
CTFHost.c:173 (High) 21:11:25:718 - SignalStackTest # Tests          : 1
CTFHost.c:173 (High) 21:11:25:718 - SignalStackTest # Pass          : 1
CTFHost.c:173 (High) 21:11:25:718 - [/VERBOSE]
```



## Summary

- Describe your remotable interface using QIDL
- Describe your classes and how to create them or run them using CIFC
- Write your unit tests using the CTF framework
- Take advantage of our easy-to-use build system to create simple Makefiles for your components