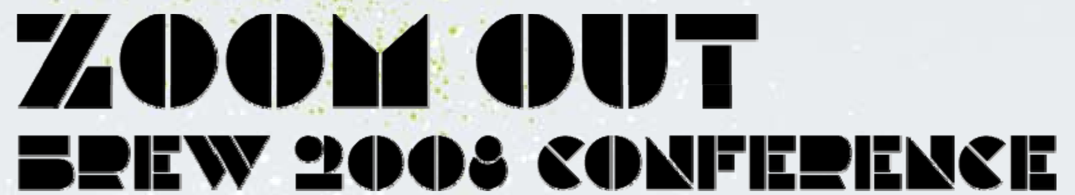


Custom OAT Development for Developers and OEMs

Mahesh “Hesh” Rohera, Senior Engineer

QUALCOMM Incorporated

QUALCOMM

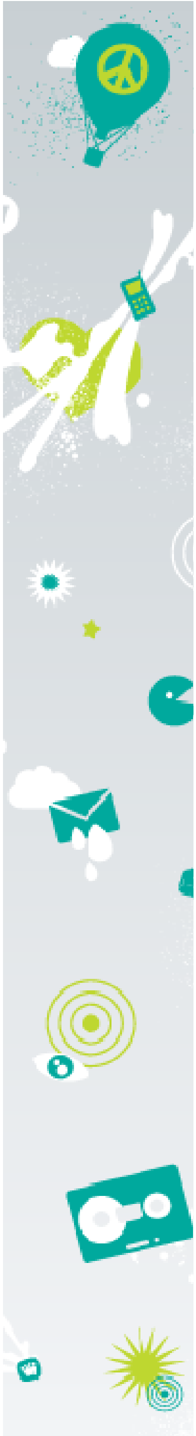
Agenda

- Need for Custom OEM Acceptance Test (OAT) Modules
- Benefits of using Custom OAT Modules
- OAT Product Kits Release Information
- Overview of OAT Architecture
- Write your own custom OAT Module
- OAT Logging Facilities
- Integrate your custom module in PEK Studio
- References

Need for Custom OAT Modules

- Standard OAT modules
  - Test standard BREW®-defined APIs
    - Behavior may be influenced by OEM-layer implementation
  - Same test cases run on all BREW handsets to ensure cross-platform conformance

- Custom OAT modules
  - Test custom-developed BREW-style extensions
  - Should be run on all platforms that expose the API
  - Should be run on all shipped builds as a regression test of the interface



Need for Custom OAT Modules

- GOAL: To ensure binary compatibility between BREW applications and all handsets exposing the API

brew 2008

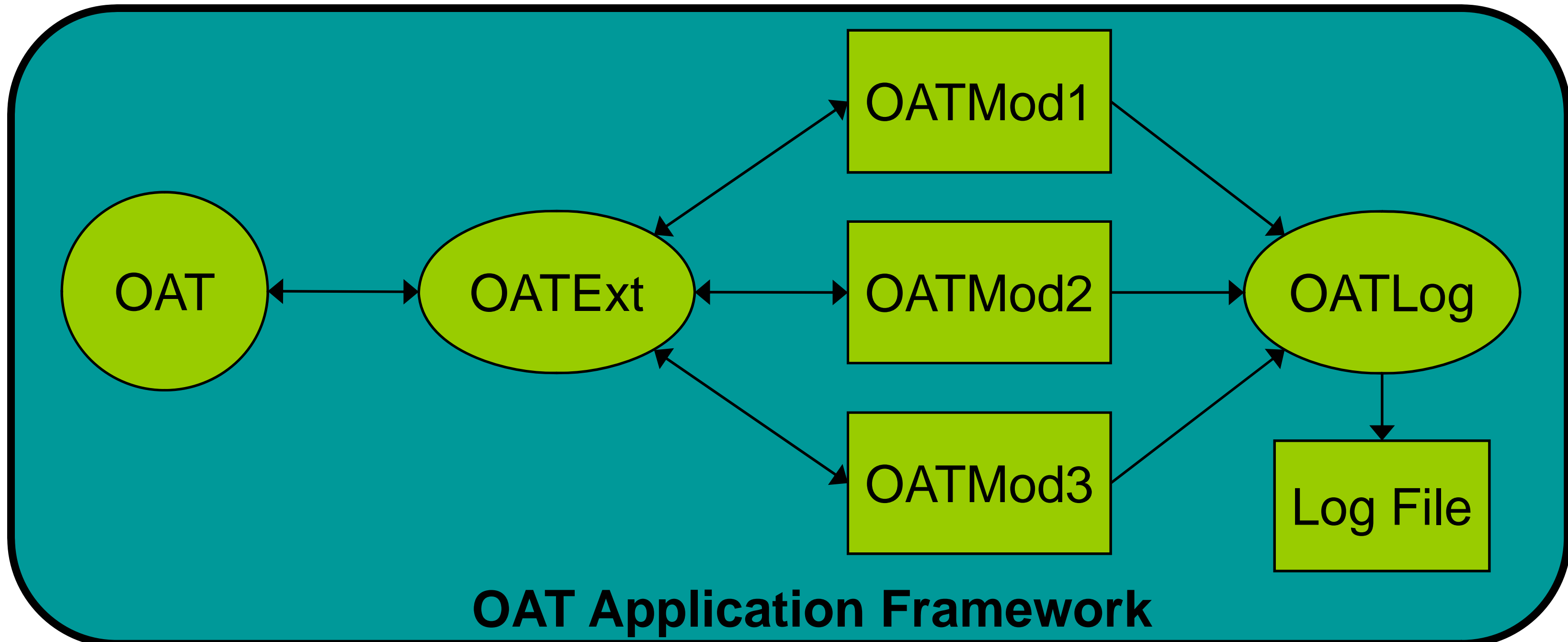
Benefits of using Custom OAT Modules

- BREW Application Developers
  - Developers who provide extensions or applications that require some form of OEM integration can benefit from implementing custom OAT modules
  - Developers can ensure that their product is ported correctly on one or more devices using the same set of custom OAT modules
- BREW Device Manufacturers
  - OEMs can use the OAT test framework to implement custom OAT modules to test their custom extensions and reuse the same custom OAT modules for testing on one or more handsets

OAT Product Kits Release Information

- BREW Application Developers
  - Developers can use the OAT Development Kit release (ODK) for developing custom OAT modules
  - The ODK enables OAT development without having access to the BREW Porting Kit (PK)
  - End users must install BREW SDK® 3.1.5SP01
- BREW Device Manufacturers
  - OEMs that produce BREW devices need to use the BREW Porting Evaluation Kit (PEK) to certify that their device is ported correctly and is compliant. Since they have access to the BREW PK, they can extend the current suite of PEK test cases with their custom OAT test cases
  - OEMs can also use the ODK to implement their custom OAT modules

Overview of OAT Architecture



- OAT is a BREW application
- OAT modules are BREW extensions
- OAT loads and runs one module at a time

brew 2008

OAT Architecture

- OAT framework defines two classes of test cases
  - “Automated”
  - “Interactive”

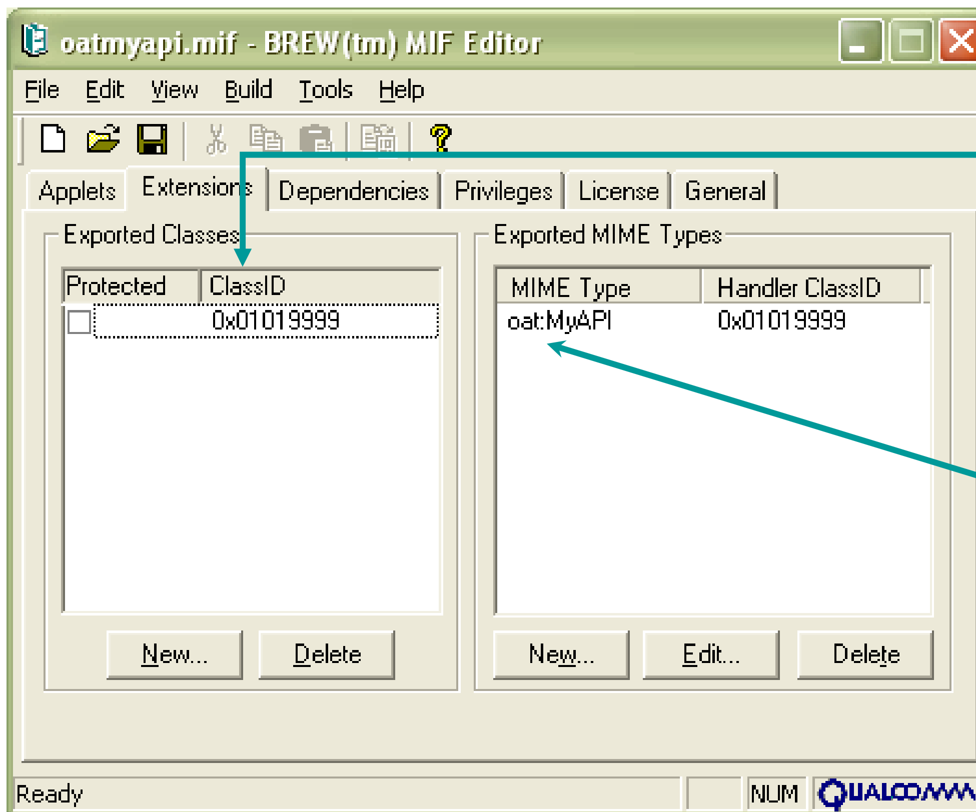
- Users choose the running mode for OAT
  - All
  - Automated
  - Interactive
  - Configured

OAT Architecture

- Test Case Results
  - OAT_SUCCESS
  - OAT_FAILED
  - OAT_UNSUPPORTED
  - OAT_POSTPROCESS
  - Deprecated
    - OAT_VERIFY
    - OAT_UNDETERMINED

Writing a new OAT Extension

OAT Module MIF File



1. Get a Class ID from Class ID generator
2. Export the Class ID
3. Export MIME Type (Base Class AEECLSID_APP)
4. Put the MIF file in the "pek\oat\src\mif" directory

Writing a new OAT Extension

- Define a data structure for your OAT test's private member variables

```
typedef struct _OATMyAPI
{
    OATExt e; // OATExt is always first

    // Class member variables can follow
    int nTestCount;
}OATMyAPI;
```

Writing a new OAT Extension

- Write the _CreateInstance() function
  - Use OATEXT_New(Ex) to initialize the module

```
boolean OATEXT_NewEx(
  int nSize,          // sizeof(OATMyAPI)
  AEECLSID cls,       // Pass directly from CreateInstance
  IShell * ps,        // Pass directly from CreateInstance
  IModule * pMod,     // Pass directly from CreateInstance
  void **ppMod,       // Pass directly from CreateInstance
  PFNSTARTTEST pfnst, // StartTest() function
  PFNENDTEST pfnend,  // EndTest() function
  PFNHANDLEEVENT pfnEvtHandler) // HandleEvent() fn
```

  - NOTE: OATEXT_New() is same as OATEXT_NewEx() except it doesn't take pfnEvtHandler.

Writing a new OAT Extension

- Write the StartTest() function

```
TestNodeList * OATMyAPICls_StartTest (
    IOATExt * pExt,
    char * name)
```

  - OAT framework will execute StartTest() whenever a module is loaded either from an initial start or a restart
  - StartTest() should MALLOC, populate and return a TestNodeList containing all the test cases in the OAT module
    - TIP: Use OATTestXXXX() macros from OATExt.h to easily populate the fields in a TestNode.
  - Copy the Module Name to the name parameter before returning

Writing a new OAT Extension

- Write the EndTest() function

```
void OATMyAPICls_EndTest(IOATExt * pExt)
```

  - OAT framework will execute EndTest() when the module exits
  - EndTest() should clean up anything that needs to be cleaned up after the test run
    - Free up memory that the test cases MALLOC'd while running
    - Release any interfaces that were created
  - NOTE: TestNodeList and TestNodes are FREE'd by the framework

Writing a new OAT Extension

- Write a HandleEvent() function (Optional)

```
boolean OATMyAPI_HandleEvent(IOATExt *pOATExt,
                             AEEEvent evt,
                             uint16 wParam,
                             uint32 dwParam)
```

  - HandleEvent() function is only required for OAT extensions that need to process BREW events (EVT_NOTIFY for example)
  - OAT Framework will forward BREW events to OATMyAPI_HandleEvent()

Writing a new OAT Extension

Implementing OAT test cases

- OATExt.h provides macros for easily declaring and registering OAT test cases in the framework
- The two types of test cases that can be registered are Automatic test cases and Interactive test cases
- OAT framework will run Interactive test cases before executing the Automated test cases

brew 2008

Writing a new OAT Extension

Automated Test Cases

- OATTestAuto
- OATTestAlwaysAuto
- OATTestAsync

Interactive Test Cases

- OATTestAlways**
- OATTestInt(Cont)**
- OATTestIntEx(Cont)**
- OATTestIntAsync(Cont)**
- OATTestIntSel
- OATTestIntAsyncEx
- OATTestStopStartBackground

** Note: Starting with PEK 3.1.5SP02 and ODK 1.0.0

- OATTestAlways is regarded as an interactive test case for purposes of reporting and user run configuration
- OATTestIntCont, OATTestIntExCont & OATTestIntAsyncCont have been introduced

OATTestAuto

- Used for test cases that can execute entirely within the test function
- OAT framework will proceed to the next test case when the test function returns

```
OATTestAuto(tn->test[tc],// TestNode
  OATMyAPI_TestAuto, // Test function
  "MyAPI_TestAuto")  // Test Name
```

OATTestAlwaysAuto

- Tests “Always” run whenever the user runs any Automated test cases
- Useful in ensuring that pre-condition test cases run before certain Automated tests
- OAT framework will proceed to the next test case when the test function returns

```
OATTestAlwaysAuto(tn->test[tc],// TestNode
  OATMyAPI_TestAlwaysAuto, // Test Fn
  "MyAPI_TestAlwaysAuto")  // Test Name
```

OATTestAsync

- Used for automated test cases that need to process callbacks, timers or events before making a Pass/Fail decision
- OAT framework will proceed to the next test case when OAT module calls IOATEXT_ResumeTest()

```
OATTestAsync(tn->test[tc], // Test Node
  OATMyAPI_TestAsync,      // Test Fn
  "MyAPI_TestAsync")       // Test Name
```

OATTestInt

- Requires the user to Pass or Fail the test
- Pass/Fail/Retry menu is provided to the user when the test function returns
- OAT Framework will automatically log the user’s decision and proceed to the next case

```
OATTestInt(tn->test[tc],  // TestNode
  OATMyAPI_TestInt, // Test Fn
  "MyAPI_TestInt")  // Test Name
```

OATTestInt

User is presented with Pass/Fail/Retry decision when the test function returns

brew 2008

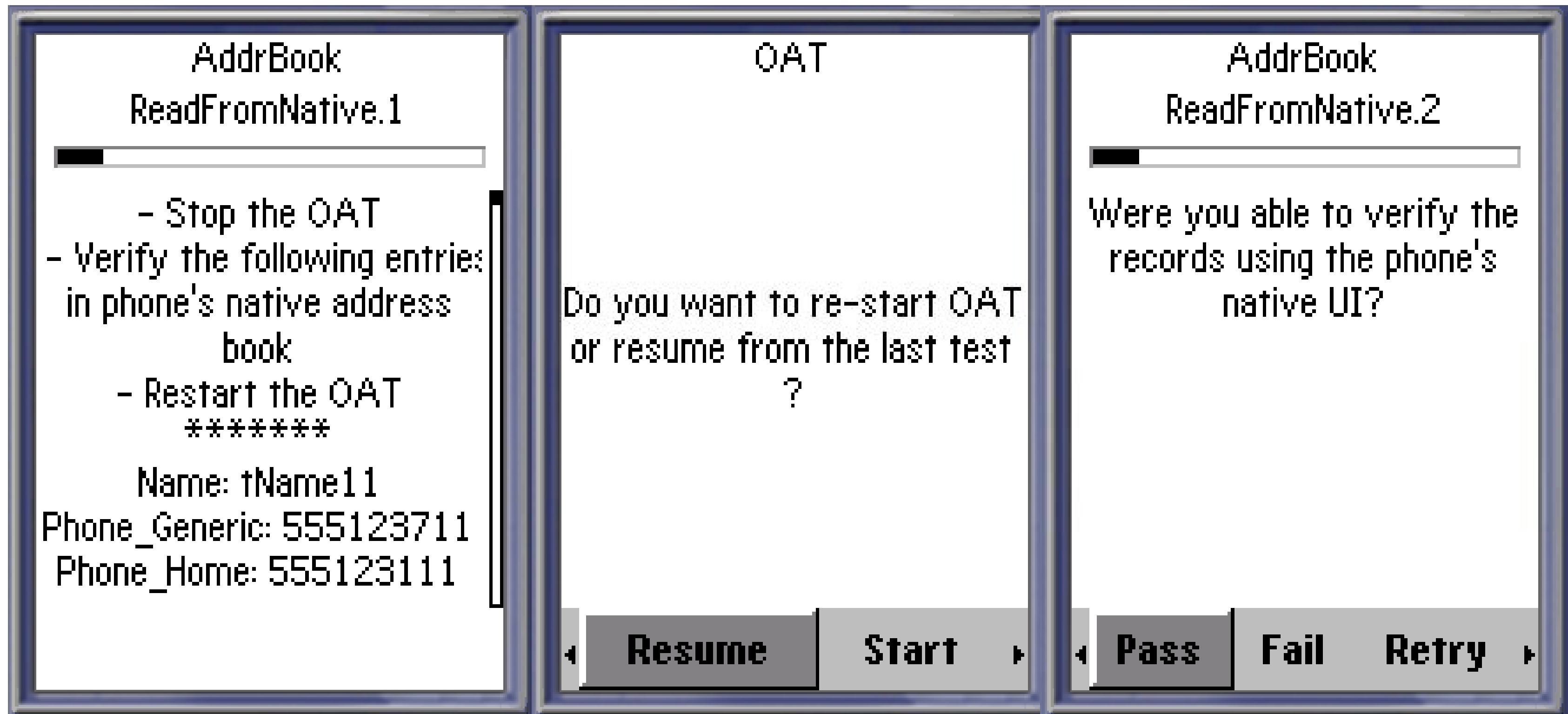
OATTestIntEx

- Requires the user to end OAT momentarily (Ex. to check the native Address Book)
- Test state is saved in OAT Framework
- When OAT resumes the user is presented with Pass/Fail/Retry decision
- OAT Framework will automatically log the user's decision and proceed to the next case

```
OATTestIntEx(tn->test[tc],  // Test Node
  OATMyAPI_TestIntEx,         // Test Fn
  OATMyAPI_TestIntEx_Resume, // Resume Fn
  "MyAPI_TestIntEx")  // Test Name
```


brew 2008

OATTestIntEx



1. Ask the user to exit OAT and do something
2. User is asked to resume the test on next launch
3. Ask the user a Pass/Fail question

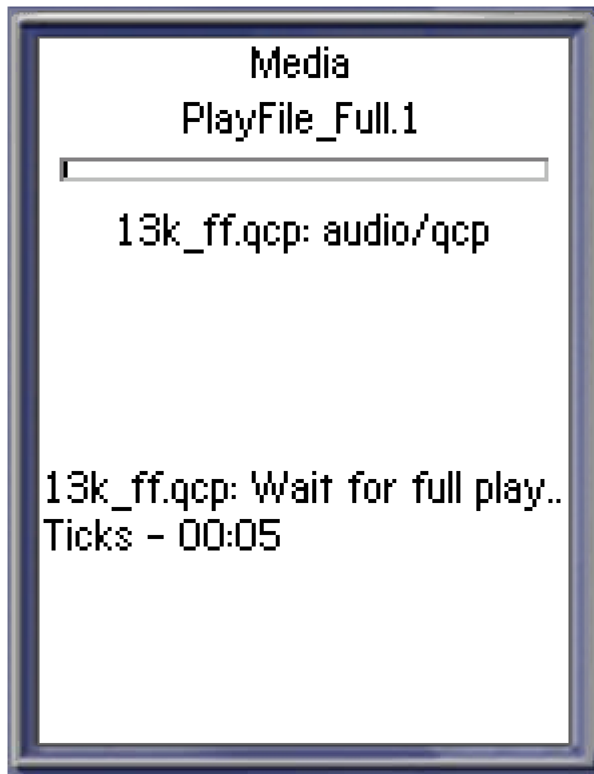
brew 2008

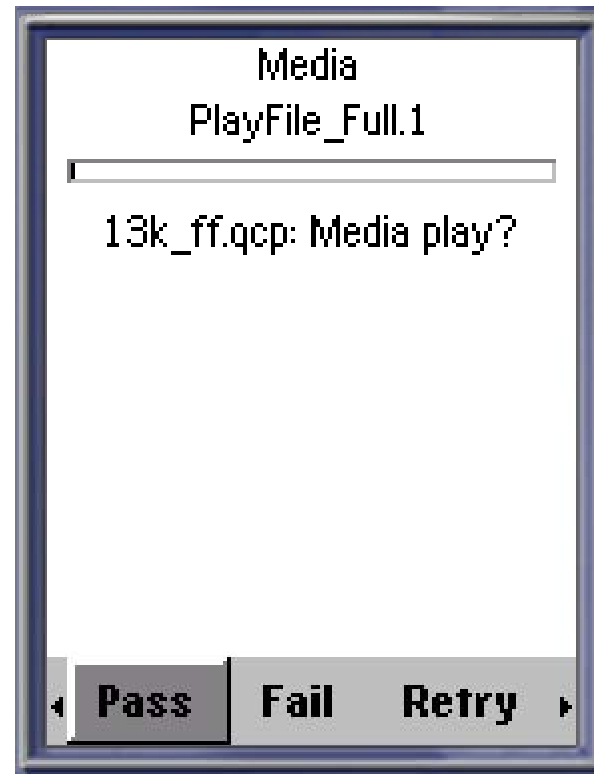
OATTestIntAsync

- Requires the user to Pass or Fail the test
- Pass/Fail/Retry menu is delayed until OAT calls IOATEXT_ResumeTest()
  - This allows the OAT Module to process events or receive callbacks before asking the user for Pass/Fail.
- OAT Framework will automatically log the user's decision and proceed to the next case

```
OATTestIntAsync (tn->test[tc], //Test Node
  OATMyAPI_TestIntAsync,  // Test Fn
  "MyAPI_TestIntAsync")   // Test Name
```

OATTestIntAsync





1. Test case starts and processes callbacks/events
2. Call IOATEXT_ResumeTest() to present the user with Pass/Fail/Retry decision

OATTestIntSel

- Requires the user to Pass or Fail the test
- Pass/Fail/Retry menu is delayed until the user presses the SELECT key
  - This allows the OAT module to use the entire display for the test case
- OAT Framework will automatically log the user's decision and proceed to the next case

```
OATTestIntSel (tn->test[tc], // Test Node
  OATMyAPI_TestIntSel,      // Test Fn
  "MyAPI_TestIntSel")       // Test Name
```

OATTestIntSel

1. Test case starts and may consume the entire display
2. User is presented with Pass/Fail/Retry decision after pressing SELECT

OATTestAsyncEx

- Automatic Pass/Fail decision
- User input might be required in order for the test to progress (to answer a privacy dialog, for example)
  - Therefore, it is considered as an interactive test case

```
OATTestAsyncEx (tn->test[tc], // Test Node
  OATMyAPI_TestAsyncEx,     // Test Fn
  "MyAPI_TestAsyncEx")      // Test Name
```

OATTestAlways

- Tests “Always” run irrespective of user configuration
- Useful in executing pre-condition test cases
- Can be either an Interactive or Auto test case
  - If implemented as an Auto test case, the framework will proceed to the next test case when the function returns
  - If implemented as an Interactive test case, the framework will automatically log the user’s Pass/Fail/Retry decision and proceed to the next case

```
OATTestAlways(tn->test[tc], // Test Node
  OATMyAPI_TestAlways,       // Test Fn
  "MyAPI_TestAlways")        // Test Name
```


brew 2008

OATTestStopStartBackground

- Used to send OAT to the background momentarily (Ex. To ask the user to perform an action)
- Module must call IOATEXT_GoBackground()
- When OAT resumes automatic Pass/Fail decision is made

```
OATTestStopStartBackground(
    tn->test[tc],                  // Test Node
    OATMyAPI_TestSSBg,             // Test Fn
    OATMyAPI_TestSSBg_Resume// Test Resume Fn
    "MyAPI_TestSSBg")              // Test Name
```

Additional Interactive Test Cases

- The following interactive test macros are similar to their counterparts (i.e. non “Cont” macros) as discussed previously:
  - OATTestIntCont
  - OATTestIntExCont
  - OATTestIntAsyncCont

- The difference is that these support a Continue/Retry dialog instead of a Pass/Fail/Retry dialog

- Useful in situations where users can be given instructions before continuing the test case. For example “Select Continue and then place a missed call on the test device.”

OAT Logging Facilities

IOATLOG (see OATLog.h)

- IOATLOG_StartEx
  - Use this at the beginning of the test function to log start of test
  - **Ex. NAME:MyAPI_TestIntAsync.1:INT:TestIntAsync**

- IOATLOG_Status2(V)
  - Logs the result of a test case, severity and optional message
  - Note: Use of IOATLOG_Status() is discouraged as it lacks support for case severity indication
  - **Ex. STATUS:FAILED:HIGH The test failed.**

- IOATLOG_WriteModVer
  - Writes the version string to the OAT Log
  - **Ex. OAT AddrBook Ver:3.1.5.33**

IOATLOG

- IOATLOG_Msg(V)
  - Logs an explanatory message
  - **Ex. MSGH: Received first expected callback**

- IOATLOG_MsgPri(V)
  - Logs an explanatory message with priority level
  - **Ex. MSGM:Failed to open media file**

OAT Logging Facilities

IOATLOG

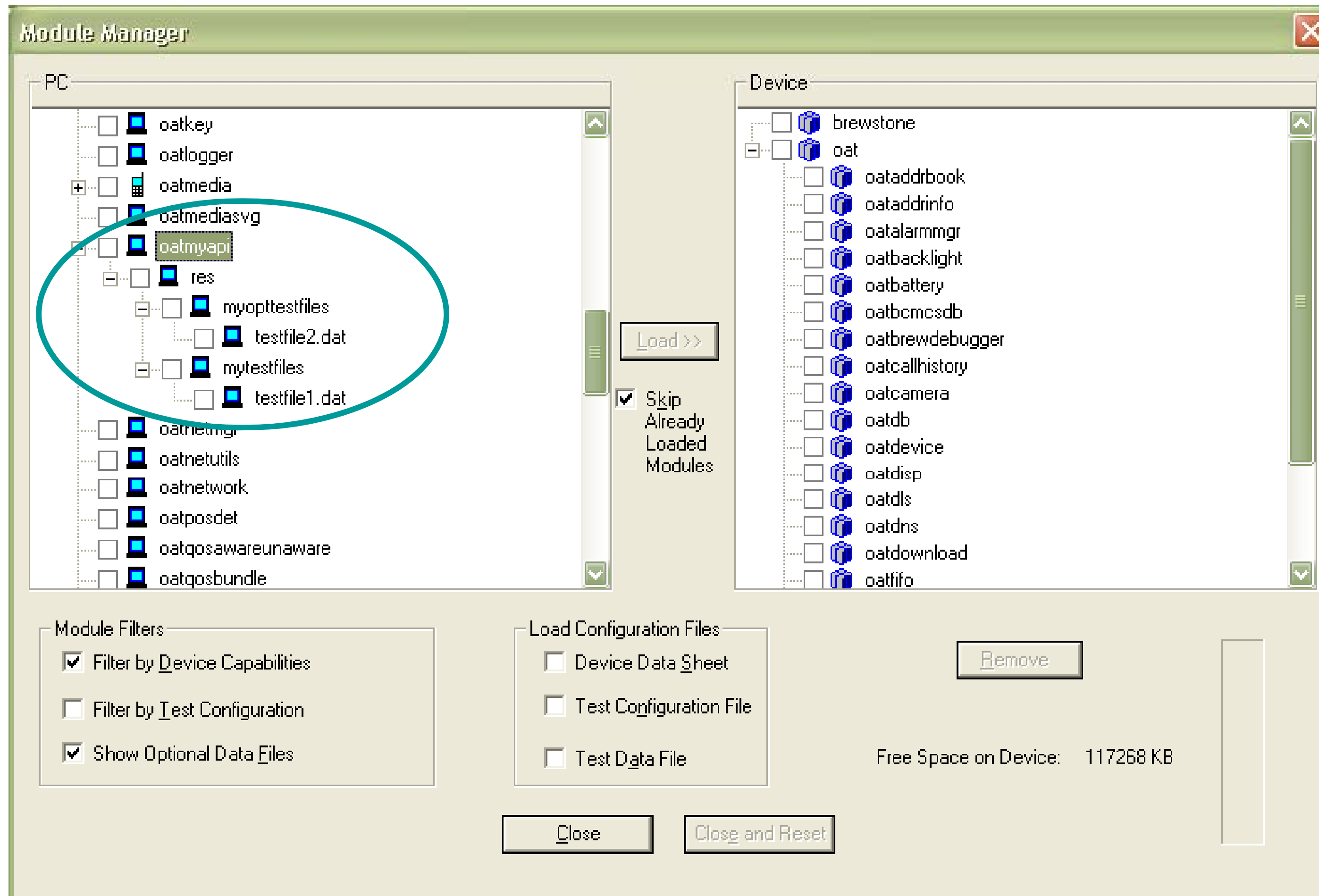
- (New helpers included in PEK 3.1.5 Releases)
  - IOATLOG_WriteString
  - IOATLOG_WriteInt
  - IOATLOG_WriteBoolean
  - IOATLOG_WriteBin
  - IOATLOG_WriteStruct
    - Logs Name-Value pairs
    - Useful when “post processing” of OAT results is required
    - **Ex. MyStruct:{1234, TRUE}**
    - **Ex. CFGI_THEMENAME:fs:/mod/19917/t1/theme.bar**

Integrating with PEK Studio

- PEK Studio automatically detects all modules in the “pek\oat\bin” directory
- TIPS
  - Create your OAT module directory in the pek\oat\src directory. Ex. pek\oat\src\oatmyapi
  - Put the MIF File in pek\oat\src\mif directory
  - Construct a makefile for your OAT module, using existing modules as examples
  - This makefile builds a .MOD file and copies it to pek\oat\bin directory

Integrating with PEK Studio

"Modules->Module Manager…" dialog



brew 2008

Integrating with PEK Studio

PEK Studio “Configure -> OAT Tests…”

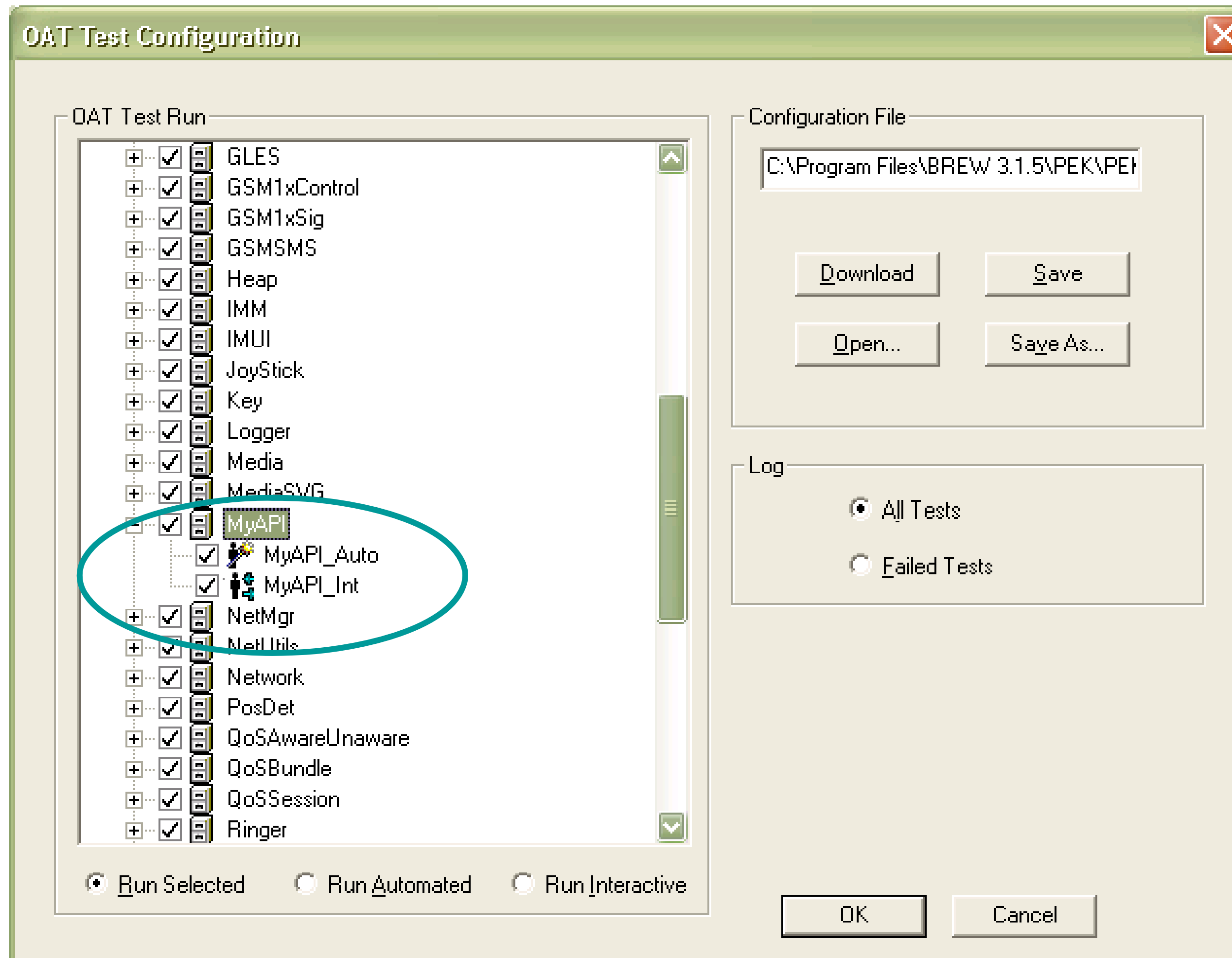
- Easier beginning in PEK 3.1.4
- Implemented by reading in testdesc.txt from each module’s directory

```
/*
{{TEST_LIST(MyAPI)
  MyAPI_Auto:auto
  MyAPI_Int:int
}}TEST_LIST(MyAPI)
*/
```

- Format is <TestName>:<int/auto>

Integrating with PEK Studio

PEK Studio “Configure -> OAT Tests” dialog



brew 2008

Adding Files to the Device File System

- Useful if your OAT module requires files be placed on the device file system

- How?
  - Beginning in PEK 3.1.5, each module has a PEKModuleData.xml file to describe additional assets to be loaded to the device

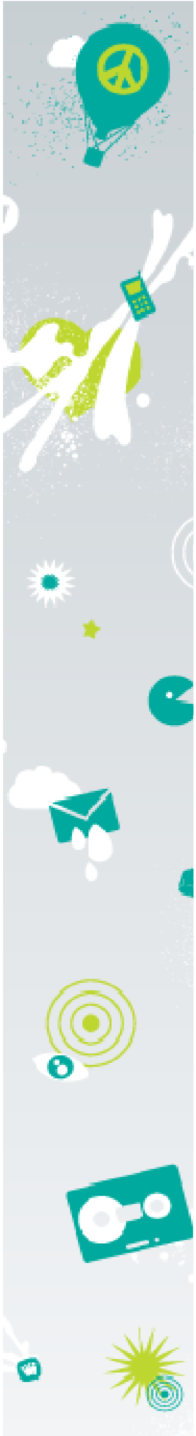
brew 2008

Integrating with PEK Studio

PEKModuleData.xml file

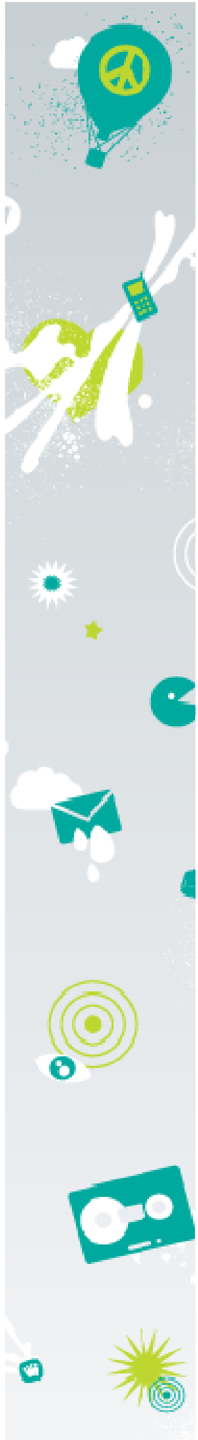
```
<PEKITEMS>
 <OATMODULE NAME="oatmyapi">
  <DEVICE_CLASSIDS>
   <CLASSID NAME="IDS_AEECLSID_LAST">
    <DATAFILES DST="$BREW_ROOT_DIR/oat/myapi"
               SRC="$PEK_ROOT\oat\src\oatmyapi\res\mytestfiles"
               TYPE="D2D" PARENT_DIRS="1"
               MANDATORY="1"></DATAFILES>
    <DATAFILES DST="$BREW_ROOT_DIR/oat/myapi"
               SRC="$PEK_ROOT\oat\src\oatmyapi\res\myopttestfiles"
               TYPE="D2D" PARENT_DIRS="1"
               MANDATORY="0"></DATAFILES>
   </CLASSID>
  </DEVICE_CLASSIDS>
 </OATMODULE>
</PEKITEMS>
```

- BREW Porting Evaluation Kit (PEK) 3.1.5 User Guide Appendix B: Creating Custom OAT Modules
- OAT Primer released in ODK 1.0.0
- Consider existing OAT modules as examples



Custom OAT Development for Developers and OEMs

Questions?



Custom OAT Development for Developers and OEMs

Thank You.