Debugging Technology Guide for Developers

brew mp™

Contents

Debugging Technology Guide for Developers

Base version:	Brew MP 1.0.2
Tested version:	Brew MP 1.0.2
Document number:	HT80-VT500-221 Rev B
Date published:	May 23, 2012

The Debugging Technology Guide provides information on the following:

- Heap debugging tools available in Brew MP
- Debugging using Heap1Wrapper
- Debugging random memory corruption
- Resolving memory leaks
- Resolving interface leaks

Sample code

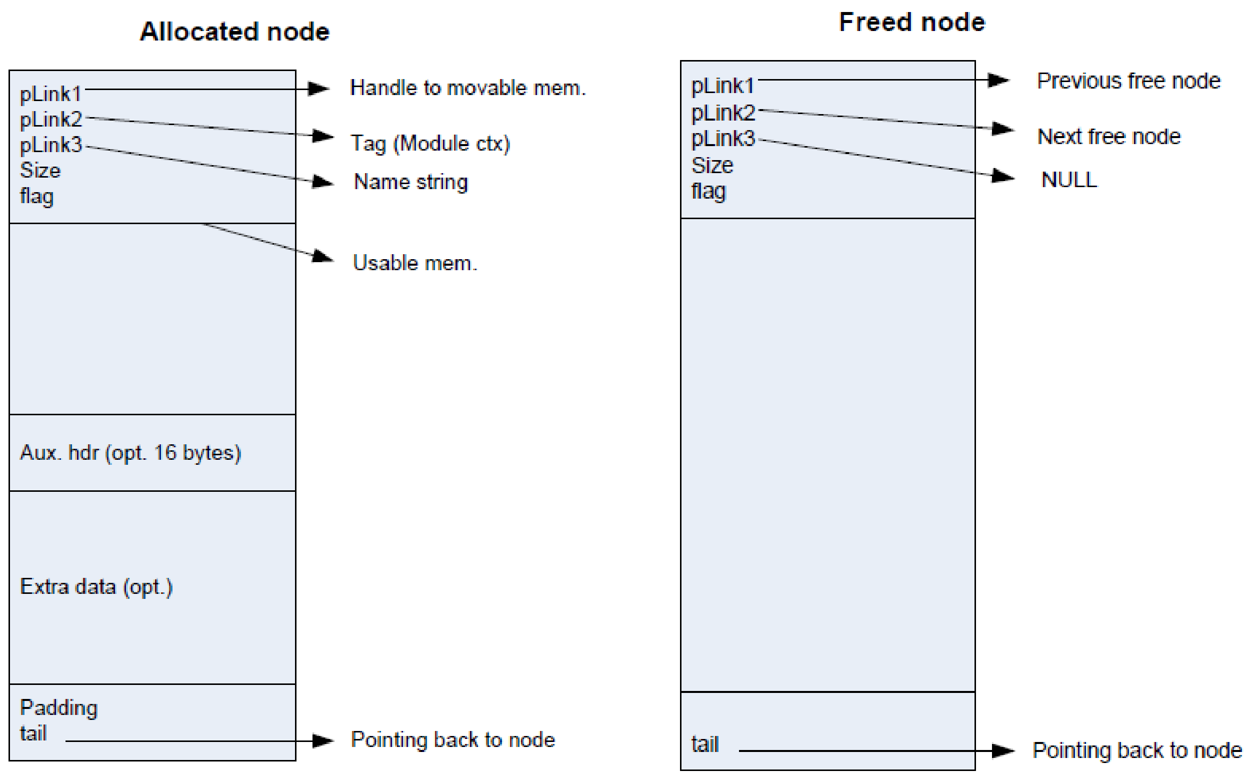
The memtests sample code is available on the Brew MP website with this guide and provides a way to produce the following types of memory errors:

- Buffer overrun or underrun
- Freeing a memory node more than once
- Writing to freed memory
- Large malloc
- Malloc memory leak
- Interface leak

ZIP filename	Location	Run app
memtests.zip	Brew MP Resources	• Download and extract the ZIP file. • Compile the extension. • Compile the app. • Run it on the Simulator.

Heap node structure

All heap nodes are packed one next to the other in heap memory. The heap node header occupies the first page of the node, and is followed by the usable memory. Depending on the state of a heap node, the content of the heap node header fields varies, as shown in the figure below:



Memory leak messages

When a module is unloaded, Brew MP frees any memory in the module context that was not released by the application (this is considered a leak). It is possible for a module to close, but not be removed from memory. Information on memory leaks is printed through the simulator message window, the Brew MP logger, or the Qualcomm eXtensible Diagnostic Monitory (QXDM) log. QXDM is an AMSS diagnostic tool that includes the ability to capture log files.

Note: Memory leaks from services running in the kernel, or more specifically, memory leaks from code that does not run in the Brew context (for example, service classes) are not detectable by Brew MP.

Memory leaks in system context or leaks in modules that are always running can exhaust all available memory. Memory leaks often cause memory fragmentation and accelerate a system crash.

When a memory leak is detected, Brew MP generates a memory leak trace message. The messages can be seen with the Brew MP Logger, QXDM, or the Simulator Log window.

```
4/7/2011 8:58:34 PM AEEModule.c:266 - Warning -- memory leak, freeing,
   0x6755760, .\hello.c:196, fs:/usermods/hello, size 100
```

Where:

- 0x6755760 - is the address of the leaked memory allocation.
- \hello.c:196 - is the file name and line number of the allocation.
- fs:/usermods/hello - is the module that leaked the memory.
- size 100 - is the size in bytes.

The leaked memory allocation is returned to the heap.

The Heap Analyzer can be used to debug memory leaks and fragmentation problems.

Getting leak notifications

Applications can register to receive notifications of leaks. To receive the notification, the application must register for the NMASK_SHELL_SYS_ERROR notification.

When a leak is detected, the OEMNTF_SYS_ERROR notification is sent. The notification data, AEESysError, contains the error information.

```
se.nType = SYSERR_MEMFREE;
se.nErr  = EALLOCATED;
se.cls   = class id of module.
```

Enabling heap debugging on the device

To enable heap debugging on the device so that the filename and line number can be displayed instead of NONAME, in the source file before including AEEStdLib.h, add the following:

- For allocations through macros (for example, MALLOC, REALLOC), define AEE_DBG_HEAP to be 1.
- For allocations through the IEnv and IRealloc interfaces, define _HEAP_DEBUG to be 1.

For example:

```
#define AEE_DBG_HEAP1
#define _HEAP_DEBUG 1
#include"AEEStdLib.h"
```

With heap debugging enabled, memory leaks are reported as follows:

```
AEEModule.c00266Warning --memory leak,freeing,0xABCD1010, .\myapp.c:189,
      size328AEEModule.
```

In newer versions of Brew MP (1.0.2.372 and above), log messages may also include the home directory of the module that caused the leak. For example:

```
AEEModule.c00266Warning --memory leak,freeing,0xABCD1010, .\myapp.c:189,fs:/mod/myapp,
      size328
```

Interface (object) leaks

Leaked objects are not directly reported, but indirectly, by analyzing memory leak reports it is possible to identify leaked objects.

A sample set of leak notifications when an IWindowMgr object is not released before an application is removed from memory:

```
4/7/2011 8:54:16 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x7370680, ModEnv, fs:/usermods/hello, size 108
4/7/2011 8:54:16 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E220, ZTARemoteObject, fs:/usermods/hello, size 28
4/7/2011 8:54:16 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E260, IWindowMgrApp1Stub, fs:/usermods/hello, size 28
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E4E0, WindowMgr1, fs:/usermods/hello, size 40
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E5E0, NONAME, fs:/usermods/hello, size 24
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E6A0, ZTARemoteObject, fs:/usermods/hello, size 28
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738E6E0, ZTARemoteObject, fs:/usermods/hello, size 28
```

```
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738EE00, src/libmod.c:118, fs:/usermods/hello, size 52
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x738EE60, NONAME, fs:/usermods/hello, size 44
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x7393760, StubIQI, fs:/usermods/hello, size 16
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x73937B0, StubIMemMap, fs:/usermods/hello, size 16
4/7/2011 8:54:17 PM AEEModule.c:266 - Warning -- memory leak, freeing,
 0x740B5E0, NONAME, fs:/usermods/hello, size 102
```

Notice that each leak notification points to files not in the user application, or have NONAME where the source file and line number should be.

For more information, see Resolving interface leaks on page 14.

Heap debugging tools

Brew MP provides the following tools that can be used for heap debugging:

- Memory analysis tools and utilities

  Includes Heap Analyzer, Heap1Wrapper , and OEMNotifyListener. These tools enable tracking of memory allocation, de-allocation, usage, and profiling, which can help detect:

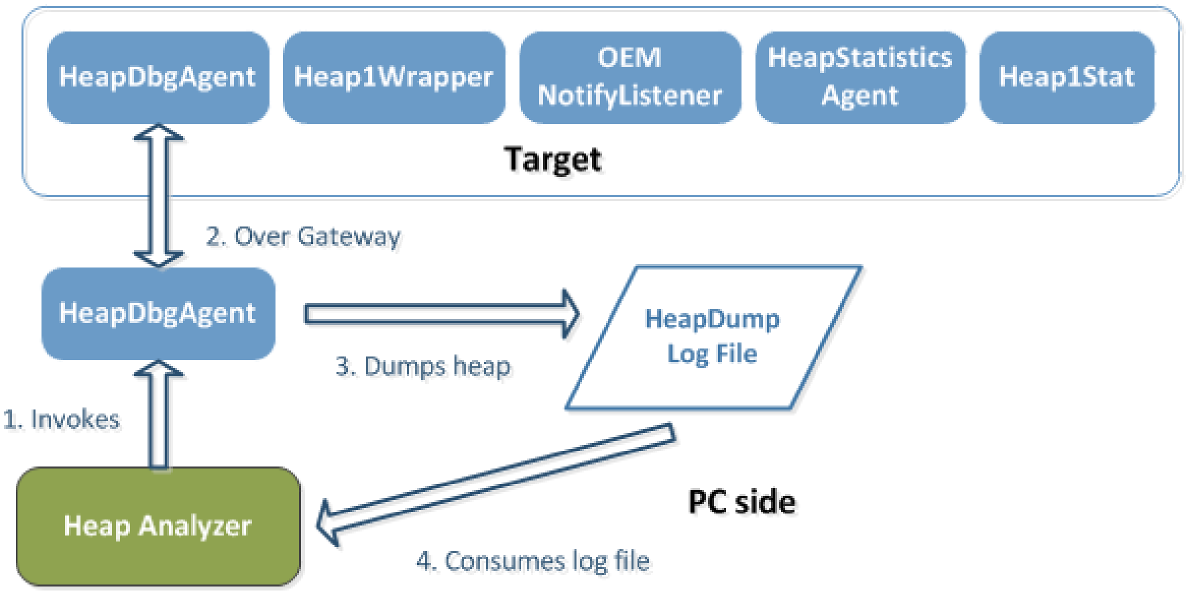
  - Potential memory leaks if memory usage continues to go up unexpectedly
  - Memory hogging modules or applets
  - The degree of fragmentation based on the maximum free heap node
  - Memory corruption

Memory analysis tools and utilities

The memory analysis tools and utilities are compatible with Brew MP 1.0.3 and 1.0.4. For Brew MP 1.0.2, Brew MP 1.0.2.549.1 or later is required. The memory analysis tools and utilities consist of the following:

- Heap Analyzer
  - A PC side .exe tool
  - Installs Heap1Wrapper, OEMNotifyListener, Heap1DbgAgent, HeapStatisticsAgent, and Heap1Stat
  - Walks the heap via interacting with Heap1DbgAgent at run time and produces similar output as heap_print.cmm

  Heap Analyzer invokes HeapDbgHelper to dump heap information, as shown in the following figure:



HeapDbgHelper is a PC Side library (DLL) that connects to the target (device or Simulator) over gateway and gets the remote object (IHeap1Walker interface).

HeapDbgAgent is a target program that exposes the IHeap1Walker interface that provides the PC side caller with the heap walk data.

HeapStatisticsAgent implements IGetHeapStatistics to get total heap and split heap statistics. It can also be launched in the background to log heap statitics on a device.

Heap1stat implements IHeap1Stat. HeapStatisticsAgent uses IHeap1Stat to split heap statistics.

For more information, see the *Tools Reference* in Resources on the Brew MP website.

- Heap1Wrapper
  - Dynamic MOD1 utility
  - The core component for memory analysis, which tracks every single memory allocation and de-allocation and inflates the heap node with the following information:
    - Current running Thread ID
    - Current running Application ID (supported in toolset version 7.11.12)
    - 64bit Timestamp
- OEMNotifyListener
  - Dynamic MOD1 utility
  - The supporting component to Heap1Wrapper, that:
    - Stores currently running Brew application ClassID
    - Keeps track of Brew application context switches
  - Enabling OEMNotifyListener requires the following, or newer, platform versions:
    - 1.0.2.641
    - 1.0.3.925
    - 1.0.4.363

    In toolset version 7.11.12, and newer, Heap1wrapper automatically enables OEMNotifyListener when loaded.

    Simulator targets always have OEMNotifyListener enabled.

Installing memory analysis tools and utilities

The memory analysis tools and utilities can be installed as follows:

1. Enable the gateway and developer mode on the device.
2. Connect the device to the PC through USB.
3. Run HeapAnalyzer.exe.

   You can run HeapAnalyzer from Target Manager by right-clicking on the target and selecting HeapAnalyzer.

   If the memory analysis utility modules have not been installed on the device, the first time Walk Heap is executed, Heap1Wrapper, OEMNotifyListener, Heap1DbgAgent, HeapStatisticsAgent, and Heap1Stat will be installed to the device.
4. Enable OEMNotifyListener.

   In the 1.0.2.641, 1.0.3.925, and 1.0.4.363 versions of Brew MP, if Heap1Wrapper is installed, OEMNotifyListener will be enabled. You can also provide your own code to set the /BREW/Shell/OEMNotifyAppCtxt settings item to enable OEMNotifyListener.
5. Configure the config.ini for Heap1Wrapper to enable the desired features.
6. Reboot the device.

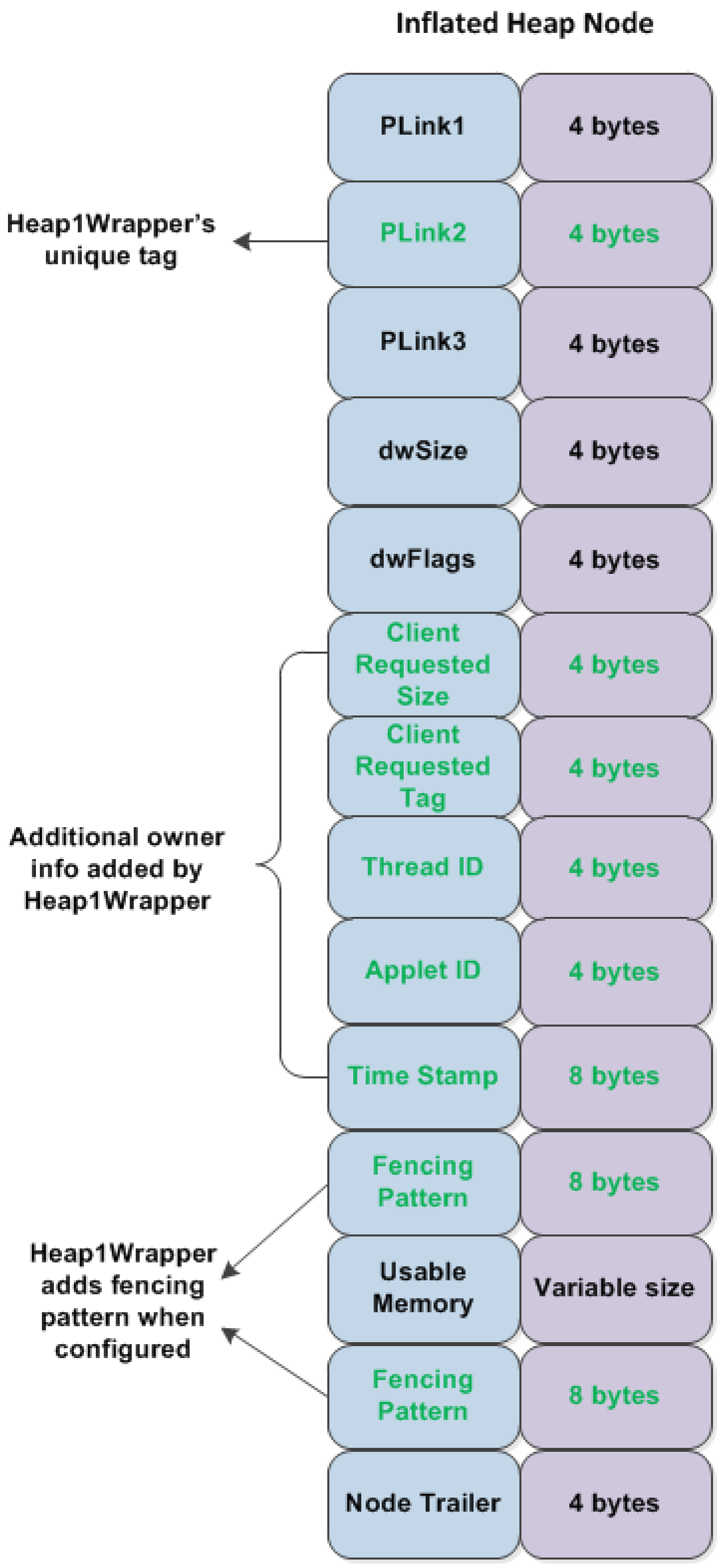
   Heap1Wrapper should be automatically started.

Heap1Wrapper

Heap1Wrapper is an in-process class that is instantiated when the system event AEEUID_SysEvt_DynMods is triggered by OS Services. Heap1Wrapper does the following:

- Replaces the vtable of the OS Services Heap with a local static vtable to handle all IRealloc methods.
- Also replaces the vtable of IHeap1Op to handle Heap1Op_Transaction methods.
- Inflates the heap node and adds owner related information.

  An inflated heap node is shown in the figure below:



- Heap1Wrapper replaces pLink2 with its unique tag (a unique memory address) to identify the node as an inflated node.

- Nodes allocated before Heap1Wrapper is instantiated are regular nodes (do not have the unique tag)
- A Fencing pattern can be added to detect buffer overrun and underrun.
- A FreeFill pattern can be applied to the node content when the node is freed to detect use after free.

For more information on using a Fencing pattern or a FreeFill pattern, see Configuring Heap1Wrapper on page 9.

Configuring Heap1Wrapper:

There are a number of Heap1Wrapper features that can be configured in the config.ini in the Heap1Wrapper directory. After you update config.ini, the device needs to be power-cycled for the new configuration to take effect.

Full node inflation

To enable full node inflation, which enlarges each heap node to add owner information, add the following line to the Heap1Wrapper config.ini file:

```
inflateHeapNode=1
```

Note: Full node inflation is required for all Heap1Wrapper features.

Node validation

Node validation validates the heap node whenever Brew MP walks the heap (for example, upon app closing) to make sure that:

- Size of each node is still 8-byte aligned
- The node trailer still points back to the beginning of the node
- Fencing pattern, if enabled, has not been overwritten

To enable node validation upon Brew heap walk, add the following line to the config.ini file:

```
enableNodeValidationOnWalk=1
```

Crash/break upon corruption

Enabling Crash/break upon corruption allows the device to crash (via data abort) or break (if a debugger such as Visual Studio is connected) when memory corruption is detected.

To enable crash/break upon corruption, add the following lines the the config.ini file:

```
debugBreak=1
debuggerOn=1      //  only need if a debugger is connected
```

Fencing pattern

This allows Heap1Wrapper to insert a fencing pattern before and after the usable memory to catch buffer overrun and underrun.

To enable the insertion of a fencing pattern, add the following lines to the config.ini file:

```
enableMemoryFence=1
memoryFenceSize=8          // specifies number of bytes, must be a multiple of 8
memoryFencePattern=0x5C
```

FreeFill pattern

If FreeFill is enabled, Heap1Wrapper fills the usable memory with the FreeFill pattern (0xFC) when a heap node is freed.

To enable the insertion of the FreeFill pattern, add the following line to the config.ini file:

```
enableFreeFill=1
```

Delayed-free

Enabling delayed-free allows Heap1Wrapper to keep the node to be freed in a queue until the queue is full, at which point, the nodes in the queue are freed in a FIFO order; enough nodes will be freed to allow new nodes to be added to the queue. Delayed-free makes it possible to detect the use of memory that has been freed.

To enable delayed-free, add the following lines to the config.ini file:

```
enableDelayedFreeHeapNodes=1
totalDelayedFreeCount=50000      // size of the delayed free queue
```

Bucketizer

This allows heap usage to be tracked on a system-wide basis across different allocation sizes. This helps systems engineers understand the heap usage trends over a period of time, say during a system-test run.

By default, two bucket sizes are defined: allocations of size 4040 bytes, and 4041 (meaning all allocations greater than this). This is especially useful in a split-heap scenario where a systems engineer wants to tune the sizes of the small heap and big heap. Enable (1) this option to keep the default sizes, otherwise Disable (0) it.

To enable bucketizer, add the following lines to the config.ini file:

```
enableMallocProfiling=1

keepDefaultBucketSize=0     //by default, two bucket sizes are defined
                            // allocations of size 4040 bytes, and 4041 (meaning
                            // all allocations greater than this).  This is useful
                            // in a split-heap scenario where a systems engineer
                            // wants to tune the sizes of the small heap and big heap.
                            // Set to 1 to keep the default sizes, otherwise set to 0.

totalBuckets=3              // If default bucket size is disabled, totalBuckets and
                            // the size ranges then need to be defined. totalBuckets,
                            // if defined, must be >= 2. In this example, there are 3
                            // buckets that cover:
                            //      1.sizes <= 4000 (bytes)
                            //      2.sizes <= 9000 && sizes > 4000
                            //      3.sizes > 9000

bucketSize1=4000
bucketSize2=9000
bucketSize3=9001            //  Value must be larger than the previous bucket size
```

Heap node debugging

This allows Heap1Wrapper to track memory allocation based on several criteria. The criteria include:

- The type of heap operation. Specify in the config.ini file for Heap1Wrapper the type of heap operation to track, which includes one or any combination of the following:

  - Malloc operation = 1
  - Realloc operation = 2
  - Free operation = 4

  For example, to debug malloc and free operations (1 + 4):

  ```
  nodeDebugType = 5
  ```

  For example, to debug all operations (1 + 2 + 4):

  ```
  nodeDebugType = 7
  ```

- The size(s) of the heap nodes. Specify in the config.ini file for Heap1Wrapper the number of node sizes and the specific sizes to track.

  For example, to track two node sizes of 1122bytes and 100 bytes:

  ```
  totalSizes=2
  size1=1122
  size2=100
  ```

- The applet ID(s) that performs the heap operations. In the config.ini for Heap1Wrapper, you need to enable ClassID mapping and then specify the number of ClassIDs and the individual ClassIDs to track:

  For example, to track ClassID 0x1001000 and 0x1880900, add the following lines to config.ini:

  ```
  enableClassIDMapping = 1
  totalClassIDs=2
  classid1=0x1001000
  classid2=0x1880900
  ```

  To use this feature, crash/break upon corruption should also be enabled, as described above. When the crash/break criteria are met, Heap1Wrapper is able to break or crash when corruption is detected to allow further debugging.

Heap statistics

Heap1Wrapper can used to continuously track the high watermark and heap usage of the kernel heap. By default, it always tracks these two numbers from the moment it is loaded at boot-up. This feature allows the data to be logged continuously to a file. The default logging interval is 1000 ms.

To enable applet and node size tracking, add the following lines to the config.ini file:

```
enablePeriodicStats=1
interval=1000            // interval in ms for logging
```

Using Brew MP debug messages on a device

Brew MP can send system state information to the diagnostics serial port. You can use the Brew MP Logger, QXDM, or the log window on the Brew MP Simulator to view the debugging messages.

To use this feature, the device build must be created with the macro FEATURE_AEE_DEBUGSET defined in OEMFeatures.h.

Turning on debug messages

You turn on the debug messages by entering a code on the device. The default code is "###BREWDEBUG#".

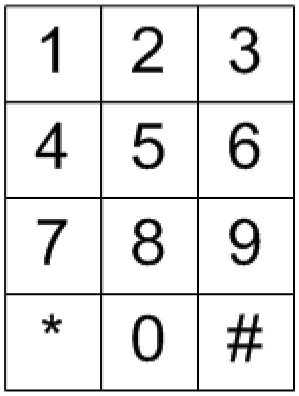
The prefix characters and termination character can be changed by the manufacturer.

To turn on debug messages:

1. On the device, go to the home screen.
2. Enter the code.

   The message "Debug Keys On" is shown at the top of the screen.

On touch screens, the screen is split into zones representing an imaginary keypad. You can enter the code by tapping in the zones representing the keys. The zones are:



1	2	3
4	5	6
7	8	9
*	0	#

You can use the QXDM keypad tool to send key events.

Enabling debug message types

You must enter key codes to enable each debug message type. The debug message types and codes are:

- Memory - "###1#"
- Heap - "###6#"
- Module information - " ###5#"
- FARF - "###324#"

There are also codes for resetting the device, "###666#", and clearing all message codes, "####".

Heap debug

You can have Brew MP dump all heap information to the serial port by entering the heap debug message code. This system walks through the heap and displays the following information:

- Total allocated heap
- Heap allocated by Brew MP
- Heap allocated by Apps
- Heap space wasted
- Total Free Heap

Module information

You can have Brew MP dump module information to the serial port by entering the module information debug message code.

The system dumps statistics about installed modules, and dumps the Brew MP registry showing the CLSID associated with a MIME type.

Debugging using Heap1Wrapper

This section provides information on using Heap1Wrapper to detect different types of memory corruption:

- Detecting double free corruption using Heap1Wrapper on page 13
- Detecting usage after free using Heap1Wrapper on page 13
- Detecting buffer overrun or underrun using Heap1Wrapper on page 14

Detecting double free corruption using Heap1Wrapper

Double free corruption occurs when an application frees the same memory more than once, which can corrupt the memory if it has already been allocated to another application. The memtests sample code, available on the Brew MP website with this guide, allows you to produce a double free error.

To detect this problem, enable delayed free in the Heap1Wrapper config.ini file by adding the following:

```
enableDelayedFreeHeapNodes=1
totalDelayedFreeCount=50000      // size of the delayed free queue
```

When a node is freed the first time, it is added to the delayed free queue instead of actually getting freed. While the node remains in the delayed free queue, if it is freed again, Heap1Wrapper reports the error in the logger or QXDM with a message similar to the following:

```
DoubleFree Err@0x22607c0:Thread Name:UI,Module Name:fs:/usermods/memtests(0x24A7EA),
      Time Stamp:17730278i
```

In this case, the message indicates that double free occurred against the buffer allocated at time 17730278, at memory address: 0x22607c0, from Applet ID 0x24A7EA, from fs:/usermods/memtests (mod or mod1) in the UI thread.

To have the device crash or break upon detection, add the following lines to the Heap1Wrapper config.ini file:

```
debugBreak=1
debuggerOn=1      //  only need if a debugger is connected
```

Detecting usage after free using Heap1Wrapper

Usage after free occurs when an application tries to read from or write to memory that has already been freed. The memtests sample code, available on the Brew MP website with this guide, allows you to produce a usage after free error.

To detect usage after free, enable delayed free and FreeFill pattern in the Heap1Wrapper config.ini file. When a node is freed the first time, the FreeFill pattern is applied to the payload of the node and the node is added to the delayed free queue instead of being freed. If the node is accessed after being freed, one of the following errors has occurred:

- Read after free - if a crash or error occurs in the application when accessing memory with FreeFill pattern (0xFC), it serves as an indication of the application trying to read data from memory that has already been freed.
- Write after free - whenever the delayed free queue is full, before a new node can be added to the queue, a node already in the queue will be removed from the queue in a FIFO order and then freed. If FreeFill pattern is enabled, before freeing the node, Heap1Wrapper will verify if the pattern is still intact. If not, it is an indication that an application has written to the memory after it was freed and it will report the error in the logger or QXDM with a message similar to the following:

  ```
  WriteAfterFree Err@0x247e220:Thread Name:UI,Module Name:fs:/usermods/memtests(0x24A7EA),
        Time Stamp:3597458
  ```

In this case, the message indicates that write after free occurred against buffer allocated at time 3597458, at memory address: 0x247e220, from Applet ID 0x24A7EA, from fs:/usermods/memtests (mod or mod1) in UI thread.

To have the device crash or break upon detection, add the following lines to the Heap1Wrapper config.ini file:

```
debugBreak=1
debuggerOn=1       //  only need if a debugger is connected
```

Detecting buffer overrun or underrun using Heap1Wrapper

Buffer overrun or underrun corruption occurs when an application writes data to a buffer and overruns or underruns the buffer‘s boundary and overwrites the adjacent memory. The memtests sample code, available on the Brew MP website with this guide, allows you to produce buffer overrun or underrun.

To detect buffer overrun or underrun, enable fencing pattern in the Heap1Wrapper config.ini file by adding the following lines:

```
enableMemoryFence=1
memoryFenceSize=8           // specifies number of bytes, must be a multiple of 8
memoryFencePattern=0x5C
```

When a node is freed, Heap1Wrapper checks to see if the fencing pattern is intact. If not, it reports the error in the logger or QXDM with a message similar to the following:

```
Heap1WrapperBufferOverrun Err@0x275d3e0:Thread Name:UI,Module
      Name:fs:/usermods/memtests(0x24A7EA),Time Stamp:3094488789i
```

In this case, the message indicates that the buffer overrun occurred against the buffer allocated at time 3094488789, at memory address: 0x275d3e0, from Applet ID 0x24A7EA, from fs:/usermods/memtests (mod or mod1) in UI thread.

To have the device crash or break upon detection, add the following lines to the Heap1Wrapper config.ini file:

```
debugBreak=1
debuggerOn=1       //  only need if a debugger is connected
```

Debugging random memory corruption

Random memory problems are the most difficult to solve. It is usually not difficult to identify a random crash, but can be difficult to determine who and when. There is no single tool that is best for locating the problem; the following can help identify the problem:

- Using the Simulator may help by providing more debug information.
- Improve code quality through static code analysis tools.
- Use a write break point if there is a data corruption problem at a specific address.

Resolving interface leaks

An interface leak occurs when the Release() method for the interface is not called before the application exits. For example, if an application invokes ISHELL_CreateInstance() or MyClass_CreateInstance(), but does not call the corresponding IInterfaceName_Release() or MyClass_Release(), this would be an interface leak. The memtests sample code, available on the Brew MP website with this guide, allows you to cause an interface leak.

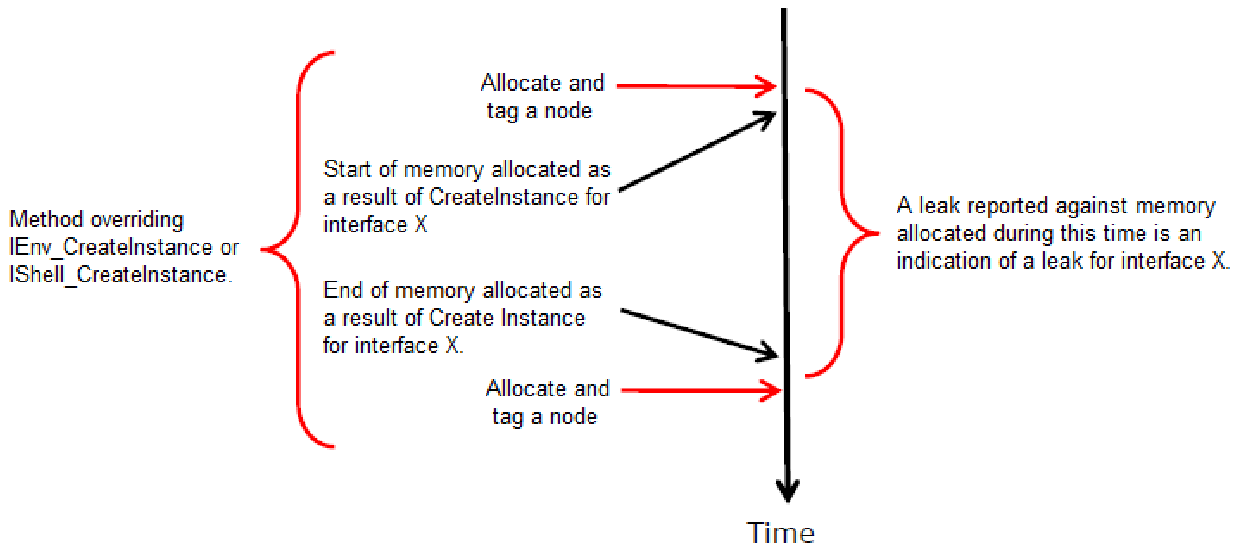
Brew MP does not call a destructor such as IInterfaceName_Release() or MyClass_Release() if the heap node is a memory allocation for a Brew MP Interface or an extension class, respectively.

Brew MP reports interface leaks rather ambiguously as a set of leaked heap allocations. These reports may or may not contain sufficient information to narrow down the specific interface that was leaked or the location in the application where the interface was created.

Solution: Override CreateInstance() methods

To determine which interface or extension class was leaked, you can override (create a wrapper function for) IShell_CreateInstance() and IEnv_CreateInstance() with methods that allocate and tag a node of memory immediately before and another immediately after creating the requested object. These two allocated and tagged nodes serve as timestamp references and any memory allocated as a result of the CreateInstance() should occur in between the timestamps of the two nodes.

If Brew MP reports a leak from the memory allocation occurring between the two timestamps, it serves as an indication of the application leaking a reference to the interface.


Method overriding IEnv_CreateInstance or IShell_CreateInstance.
Allocate and tag a node
Start of memory allocated as a result of CreateInstance for interface X
End of memory allocated as a result of Create Instance for interface X.
Allocate and tag a node
A leak reported against memory allocated during this time is an indication of a leak for interface X.
Time

Memory leaks caused directly by the application under test may be identified by the specific filename and line number reported in the messages. This identifies the location in the code from where the allocation was made. Interface leaks, however, are not as clearly identified.

For in-process classes, memory allocated during construction of an interface, and memory that is allocated and retained by the object as a result of API calls, is tagged as being allocated by the application that creates the interface. If the application fails to release the interface before closing, each of those allocations is reported as a leak from the application. The presence of interface leaks may be recognized by the presence of leaked nodes tagged as NONAME, ModEnv, or an unrecognizable filename and line number.

Using this solution to resolve interface leaks includes the following steps, which are explained in detail in this section:

1. Identify and correct all memory leaks from the application (i.e., resolve all the leaks wherein Brew MP reports a specific filename and line number in your application).
2. Set up HeapAnalyzer to work with your target.

3. Set up your project to use the CreateInstance() override.
4. Run your application through the use case that produces interface leaks.
5. Before exiting your application, use the Walk Heap feature of HeapAnalyzer to capture a dump of the heap nodes.
6. Exit your application, and save the memory leak messages.
7. Analyze the memory leak messages and heap dump to determine which interfaces were leaked and from where.

Requirements for the solution

This solution requires the following:

- Brew MP 1.0.2 or later
- For physical targets, the Developer Mode USB Port Setting must be set such that the Gateway is enabled. This is required for HeapAnalyzer to work properly.
- Memory allocations must be tagged. Ensure that the following preprocessor definitions are defined for the debug variants of your project:
    - AEE_DBG_HEAP=1: Enables tagging of memory allocations made via the AEEStdLib.h macros (MALLOC, REALLOC, etc.).
    - _HEAP_DEBUG=1: Enables tagging of memory allocations made via IEnv and IRealloc APIs (IEnv_ErrRealloc(), IRealloc_ErrRealloc(), etc.).

  This solutuion works best on simulated targets.
- Set up Heap Analyzer and other tools as described in Installing memory analysis tools and utilities on page 7.

Identifying and resolving memory leaks in the application

1. Run your application through the scenario that produces memory leaks.
2. Close your application.

   Brew MP will report all unreleased memory that was allocated by or on behalf of your application.
3. For each message where a file from your application was printed in the log messages, use the reported filename and line number to identify the location in the application where the memory allocation was made.

   Inspect and debug the code to determine the reason the allocation was not released, and resolve that defect. For example, if the following messages are reported:

```
 5/16/2011 1:42:03 PM AEEModule.c:266 - Warning
-- memory leak, freeing, 0x6E2E760, .\AppUtils.c:325,
fs:/usermods/ImageApp, size 20
 5/16/2011 1:42:03 PM AEEModule.c:266 - Warning
-- memory leak, freeing, 0x6EC0D00, NONAME, fs:/usermods/ImageApp,
size 64
```

   The first message indicates a memory leak from the application, as it refers to a filename from the application under test. The message indicates that the allocation made on line 325 of AppUtils.c was not freed by the application. The second message does not contain useful information and is most likely indicative of a leaked interface.

Setting up your project to override CreateInstance()

1. Back up the platform header files that are modified during this process.

   For the platform with which your project is built, locate and back up the following files, which will be modified for this override solution:

- For Brew MP 1.0.2, copy AEEShell.h to a new name, such as AEEShell_saved.h. For Brew MP 1.0.3 and later, make a copy of AEEIShell.h.
- Copy AEEIEnv.h to a new name, such as AEEIEnv_saved.h.

2. To detect leaks from interfaces instantiated from ISHELL_CreateInstance(), make the following override changes to AEEShell.h (or AEEIShell.h) to override IShell_CreateInstance().
   - Locate the beginning of the documentation block in this file and add the following:

```
#ifdef ISHELL_OVERRIDE
#include "OverrideIShellCreateInstance.h"
#endif
```

   - For Brew MP 1.0.2, locate the following declaration of ISHELL_CreateInstance() in AEEShell.h:

```
#define ISHELL_CreateInstance(p,id,ppo)
            GET_PVTBL(p,IShell)->CreateInstance(p,id,ppo)
```

   and replace it with the following:

```
#ifdef ISHELL_OVERRIDE
#define BYPASS_ISHELL_CreateInstance(p,id,ppo)
            GET_PVTBL(p,IShell)->CreateInstance(p,id,ppo)
#else
#define ISHELL_CreateInstance(p,id,ppo)
            GET_PVTBL(p,IShell)->CreateInstance(p,id,ppo)
#endif
```

   - For Brew MP 1.0.3 and later, locate the following declaration of IShell_CreateInstance() in AEEIShell.h:

```
static __inline int IShell_CreateInstance(IShell *pif, AEECLSID cls,
         void **ppobj)
{
   return AEEGETPVTBL(pif,IShell)->CreateInstance(pif, cls, ppobj);
}
```

   and replace it with the following:

```
#ifdef ISHELL_OVERRIDE
static __inline int BYPASS_ISHELL_CreateInstance(IShell *pif, AEECLSID cls,
      void **ppobj)
#else
static __inline int IShell_CreateInstance(IShell *pif, AEECLSID cls,
      void **ppobj)
#endif
{
   return AEEGETPVTBL(pif,IShell)->CreateInstance(pif, cls, ppobj);
}
```

3. To detect leaks from interfaces instantiated from IEnv_CreateInstance(), make the following override changes to AEEIEnv.h to override IEnv_CreateInstance():
   - Locate the beginning of the documentation block and add the following:

```
#ifdef IENV_OVERRIDE
#include "OverrideIEnvCreateInstance.h"
#endif
```

   - Locate the following declaration of IEnv_CreateInstance():

```
static __inline int IEnv_CreateInstance(IEnv* pif, AEECLSID cls, void** ppo)
{
   return AEEGETPVTBL(pif,IEnv)->CreateInstance(pif, cls, ppo);
}
```

   and replace it with the following:

```
#ifdef IENV_OVERRIDE
static __inline int BYPASS_IEnv_CreateInstance(IEnv* pif, AEECLSID cls,
          void** ppo)
#else
static __inline int IEnv_CreateInstance(IEnv* pif, AEECLSID cls, void** ppo)
#endif
{
   return AEEGETPVTBL(pif,IEnv)->CreateInstance(pif, cls, ppo);
}
```

4. Copy the provided OverrideCreateInstance.c, OverrideIShellCreateInstance.h, and OverrideIEnvCreateInstance.h files to your project directory.
5. Add the provided OverrideCreateInstance.c file to your project.
6. Add the following function call to the application's _InitAppData() (for MOD) or _CtorZ() (for MOD1) function before any calls to ISHELL_CreateInstance are made:

```
#ifdef ISHELL_OVERRIDE
   initOverrideCreateInstance(pMe->m_pIShell);
#endif
```

7. Add the following function call to the application's _FreeAppData() (for MOD) or _Dtor (for MOD1) function after all the calls to _Release() are made:

```
#ifdef ISHELL_OVERRIDE
    freeOverrideCreateInstance();
#endif
```

8. Define one or more of the following in the preprocessor definitions for your project:
   - ISHELL_OVERRIDE: To enable the override for ISHELL_CreateInstance().
   - IENV_OVERRIDE: To enable the override for IEnv_CreateInstance().
   - OVERRIDE_MOD1: To enable any override for MOD1 applications. Do not define this for MOD applications.

Running the application

Using the sample code provided, for every CreateInstance, the beginning node is always tagged as "CI|Start" and the ending node is always tagged with "CI|xxxxx", where xxxxxx includes the following information:

- Symbolic name of the class that is being instantiated
- 32bit ID in hex for the class
- The file name and line number where the class is instantiated
- Raw interface pointer value

To run your application:

1. Rebuild your application with the CreateIntance() override.
2. Install the rebuilt application to the target.
3. Run the steps that produce memory leaks.
   Note: Do not exit the application yet.

Walking the heap

1. Launch HeapAnalyzer.
2. Select the target on which you are testing.
3. Click the Walk Heap button to save the heap dump.

   Make note of the CSV file to which the heap dump is saved.

Exiting the application

1. Exit the application.

   Brew MP should report a set of memory leaks.

2. Save the memory leak messages to a text file.

Analyzing the leak messages and the heap dump

1. Open the heap dump CSV file in Microsoft Excel.
2. Set up filters on the data and filter the data to only show the nodes in which the Owner field is set to the application under test.
3. Sort the data smallest to largest based on the ts (timestamp) field.
4. For each leaked heap allocation reported as leaked by Brew MP, search for the address of the allocation in the sorted heap dump.

   If the address shows up between tagged allocations of the following format, that is an indication that an object of the indicated ClassID has leaked from the location shown. For example, if the sorted heap dump shows the following tags:

Address	pLink3
0x0721D4A0	CI | Start
0x0721D800	ZTARemoteObject
0x071B3220	NONAME
0x071B3220	CI|0x1072401:0x1072401|.\AppUtils.c:379|0x71b3240|

   and the following heap allocation was reported as leaked by Brew MP when the application was closed:

```
5/16/2011 4:50:32 PM AEEModule.c:266 - Warning
-- memory leak, freeing, 0x71B3220, NONAME, fs:/usermods/ImageApp, size 36
```

   This is an indication that an object of ClassID 0x1072401 (AEECLSID_CFileSystem2), which was created on line 379 of AppUtils.c, has been leaked. Further inspection and debugging of the application may be done to determine the reason this interface was not released and to fix the problem.

Note: After the memory and interface leaks are resolved, be sure to revert the changes made to AEEShell.h, AEEIShell.h, and AEEIEnv.h, and remove the ISHELL_OVERRIDE, IENV_OVERRIDE, and/or OVERRIDE_MOD1 preprocessor definitions from the application project file.

Creating a heap plug-in for debugging

Brew MP builds support a plug-in interface for debugging. Manufacturers can implement the plug-in interface to find and maintain information about the heap node. The plug-in could track the call stack, time stamp, or thread ID of the heap node.

There is no need for special builds. All debug features are selectable at runtime.

The plug-in uses the IHeap, IHeapDebugCtrl, and IHeapDebuggerinterfaces.

Creating a heap debug plug-in

To create a heap debug plug-in:

1. Create a debugger plug-in class.

   Implement the IHeapDebugger interface, defined in AEEIHeapDebugger.h.
2. Create a device class that does the following:
   - Creates an instance of the debugger plug-in class implemented in step 1.

- Creates an instance of the IRealloc interface for this environment.

```
IEnv_CreateInstance(piEnv,AEECLSID_Realloc,(void **)&piRealloc);
```

- Queries the IHeap1 interface by calling IRealloc_QueryInterface():

```
IRealloc_QueryInterface(piRealloc,AEEIID_IHeap1,
                       (void **)&piHeap1);
```

- Queries the IHeap1DebugCtl interface by calling IHeap1_QueryInterface():

```
IHeap1_QueryInterface(piHeap1,AEEIID_IHeap1DebugCtl,
                     (void **)&piCtl);
```

- Attaches the IHeap1Debugger to the Heap by calling IHeap1DebugCtl_AddDebugger():

```
IHeap1DebugCtl_AddDebugger(piCtl, piDebugger);
```

3. Create the service object, in the application or in the system context.

Heap plug-in use

When the heap debug plug-in is running, the heap invokes IHeapDebugger methods before critical operations. For example:

- IHeap1Debugger_ErrorReport() is invoked when the heap detects a program error.
- IHeap1Debugger_BeforeFree() is invoked before heap frees a node.
- IHeap1Debugger_BeforeRealloc() is invoked before heap reallocates a node.