



Binary Runtime Environment for Wireless™

BREW™ OEM Reference Guide for BRIDLE



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

This manual was written for use with the BREW SDK™ for Windows, software version 3.1. This manual and the BREW SDK software described in it are copyrighted, with all rights reserved. This manual and the BREW SDK software may not be copied, except as otherwise provided in your software license or as expressly permitted in writing by QUALCOMM Incorporated.

Copyright © 2004 QUALCOMM Incorporated
All Rights Reserved

Printed in the United States of America

All data and information contained in or disclosed by this document are confidential and proprietary information of QUALCOMM Incorporated, and all rights therein are expressly reserved. By accepting this material, the recipient agrees that this material and the information contained therein are held in confidence and in trust and will not be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without the express written permission of QUALCOMM Incorporated.

Export of this technology may be controlled by the United States Government. Diversion contrary to U.S. law prohibited.

Binary Runtime Environment for Wireless, BREW, BREW SDK, BREWStone, Mobile Station Modem, MSM, MobileShop, gpsOne, Eudora, Compact Media Extension, CMX, and PureVoice are trademarks of QUALCOMM Incorporated.

QUALCOMM, TRUE BREW, The Grinder, and QChat are registered trademarks of QUALCOMM Incorporated.

All trademarks and registered trademarks referenced herein are the property of their respective owners.

BREW™ OEM Reference Guide for BRIDLE

80-D4690-1 Rev. B

May 18, 2004

Contents

Introduction 4

In this document 4

Best Practices for OEMs 5

Enforcing the safeguards 6

Memory Protection 7

Memory regions 7

Scatter Load 7

Context Switches 10

Supervisor-to-User mode 10

User-to-Supervisor mode 11

Boundaries 12

Type II Interfaces 13

Thin AEE 13

Thick AEE 14

OEM API Design 14

BRIDLE_SWI 15

BRIDLE Macros 17

BRIDLEX_*() 17

PACKBRIDLEX_*() 18

packbridle() inline 19

UNPACKBRIDLEX_*() 20

UNPACKBRIDLE_PARAM_X 21

BRIDLE_CHECK_* 22

BRIDLE_COPY_*_USER 23

BRIDLE_SUBSYS_* 23

Examples 25

OEMFoo.h	25
BRIDGEInit.h (or BRIDGEInit_OEM.h)	25
BRIDGEInit.c (or BRIDGEInit_OEM.c)	25
bridge_foo.c	26
bridge_foo.h	26
AEEFoo.c	27
Complex parameter checking	28
Callbacks	29
Static Callbacks	29
AEECallbacks	30
Handles	31
Scatter Load	33

Other Considerations 34

Supervisor mode callback mechanisms	34
Locking interrupts	34
Interrupt service routines	35
Static applications	35
Code review	35
OAT Tests	36

Introduction

This document presents the concepts of the BREW Isolated Domain for Legitimate Execution (BRIDLE), and provides instructions for and examples of using BRIDLE to protect BREW from defective or malicious applications.

In this document

The remainder of this document contains the following sections.

Section	Description
Best Practices for OEMs	Gives recommendations that will help you get the maximum protection from BRIDLE.
Memory Protection	Discusses memory protection in the context of various hardware environments.
Context Switches	Explains the process of switching between Supervisor and User modes.
Boundaries	Explains the four types of interfaces, and how BRIDLE boundaries are used in each.
BRIDLE_SWI	Explains the BRIDLE-SWI function.
BRIDLE Macros	Provides macros that you can use to switch between Supervisor and User modes.
Examples	Provides example “BRIDLEing” sessions
Other Considerations	Discusses miscellaneous issues that you may encounter in the BRIDLE domain.

Best Practices for OEMs

This section contains recommendations for ensuring that BRIDLE is able to perform its most important job -- protecting the system from corruption by ensuring that software executing in User mode does not enter into in Supervisor mode.

Memory protection provides a defense against the most direct attempts to corrupt the system. Any BRIDLEd interface, however, provides a gateway into Supervisor mode and therefore presents a potential risk to the security of the system. The key to maintaining security is ensuring that no sequence of User mode actions can result in corruption. For example, a BRIDLEd function for reading from sockets could be misused to overwrite system memory unless appropriate precautions are taken in the BRIDLE layer.

NOTE: The initial implementation of BRIDLE will halt execution when violations are detected. This does not guarantee that Supervisor mode software will continue to execute normally, but it does protect against corruption.

Following are some examples of some precautions that should be taken to protect the system.

- Every buffer passed across the Supervisor mode boundary must be validated (at least) once in Supervisor mode. Memory that will be written or read should be checked against the corresponding permissions that apply to User mode for that range of memory.
- Refrain from passing invalid arguments to system-level software, unless it is clear that the system-level software is safeguarded against invalid values. For example, if a BRIDLEd function accepts a socket descriptor from User mode and then hands it directly to the lower layer, it would allow User mode software to pass invalid socket descriptors to system functions, which may or may not be a problem depending upon the system software implementation. Also, depending on the implementation, it could allow reading or writing from a socket the User mode task does not own.
- Once in Supervisor mode, do not trust any values that are stored in User-writable memory. In particular, any pointer read from that memory is particularly dangerous. For example:

- Interfaces stored in User mode memory should not be relied upon when in Supervisor mode. This would leave an opening for User mode code to "hijack" the Supervisor mode code by modifying the interface method pointers.
- Linked lists stored in User mode memory could be corrupted by User mode. Supervisor mode using such a linked list could loop infinitely, overrun buffers, or perform any number of unpredictable actions.

Enforcing the safeguards

In general, the easiest way to enforce the safeguards described above is to make use of handles or descriptors to identify system resources to User mode software, and storing the corresponding structures and any related pointer entirely in Supervisor mode. The handle, a small integer, is easily validated against a range after the transition to Supervisor mode.

Keep in mind that this list is not a complete prescription for a secure implementation, and that the complexity and difficulty of proper BRIDLEing will vary with the complexity of the underlying system software. For example, if complex processing is performed in the system layer, and the system layer does not provide robust validation of arguments, then the BRIDLE layer may have to perform very complex argument validation before calling the system software. Similarly, if an unexpected sequence of function calls could present a problem for the system software, the BRIDLE layer will have to take whatever steps necessary to ensure the appropriate sequencing.

Another thing to remember is that the potential for bugs in the underlying system software presents an unavoidable vulnerability. For example, if something as complex as PNG decoding is performed in the system layer, there is the possibility for certain malformed PNG files to trigger a buffer overrun or some other error that would corrupt the system. This kind of error is not something that could be anticipated at the BRIDLE layer, and even if it were anticipated the required safeguards would be unreasonably complicated and slow. The most important guideline to apply here is to keep as much complexity as possible out of the system layer, and perform that work in User mode instead.

Implementing other precautions not discussed in this document, such as employing safe programming practices, audits, and adversarial testing, is also highly advisable.

Memory Protection

The mechanism used to implement memory protection may vary depending on the hardware. For example, the ARM9 core contains a Memory Management Unit (MMU), while the ARM7 core utilizes a Memory Protection Unit (MPU), and other devices may use yet other facilities. For the purposes of this document, MMU will be used generically.

Memory regions

When the CPU needs to access a particular location in RAM, the request is first verified by the MMU, based on the current operational mode. An access violation will result in a data abort. In BRIDLE, there are five distinct memory regions defined:

1. Supervisor Read-only, User No-access:
This region is used for system code and const data.
2. Supervisor Read-write, User No-access:
This region is used for system data and ZI (zero-initialized data, or bss).
3. Supervisor Read-write, User Read-only:
This region is used for a small subset of system data that needs to be read by user mode. In particular, the module SWI numbers are required as arguments to the BRIDLE SWI in user mode, but are owned by supervisor mode.
4. Supervisor Read-only, User Read-only:
This region is used for user code and const data.
5. Supervisor Read-write, User Read-Write:
This region is used for user data and ZI.

Scatter Load

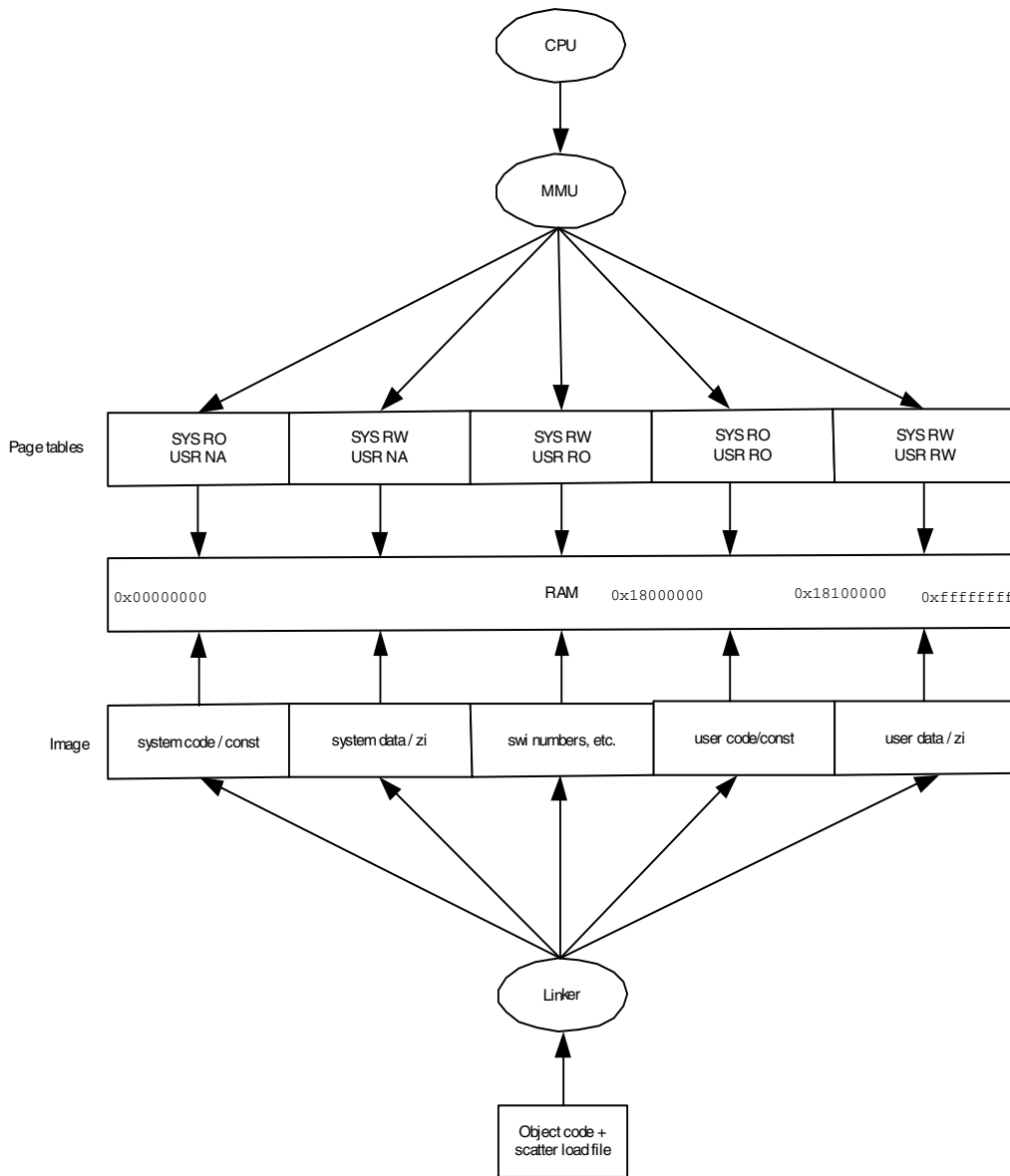
What is described in this section is ARM specific, but most software architectures will have a similar mechanism.

How the image gets loaded into RAM is specified by the ELF file. The ELF file is constructed by the linker using two inputs: the object code (i.e. libraries and object files) and the scatter load description file.

While the syntax of the scatter load file is beyond the scope of this document, it is important to understand that it is a collection of rules that describe what portions of object code end up in what regions of RAM. For example, there is a rule that says that all AEE library code/const is to be loaded into the Supervisor Read-only, User Read-only section of RAM. Additionally, there are special section names that can be used to place just a portion of a file in a different section (see examples).

The following figure illustrates the relationship between the MMU and the scatter load.

Relationship between MMU and Scatter Load



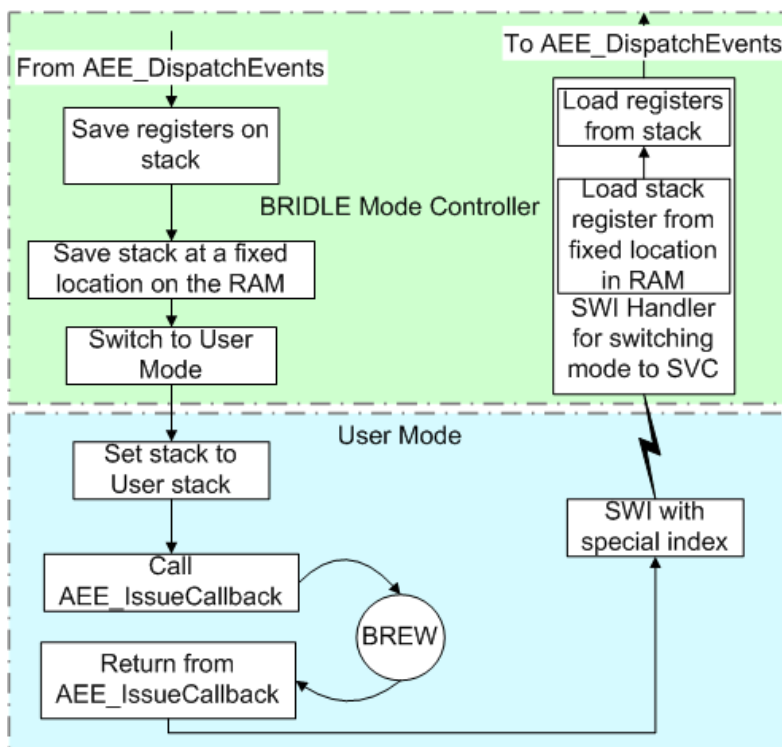
Context Switches

BRIDLE requires that a given thread of execution be able to switch between User and Supervisor modes.

Supervisor-to-User mode

Switching from Supervisor mode to User mode is safe and relatively simple. On most hardware, this simply involves setting some bits in a register. In BRIDLE phase I, this action is performed during the dispatching of callbacks, so all events are processed in User mode. The next figure illustrates the transition from Supervisor to User mode, and the reverse during function return.

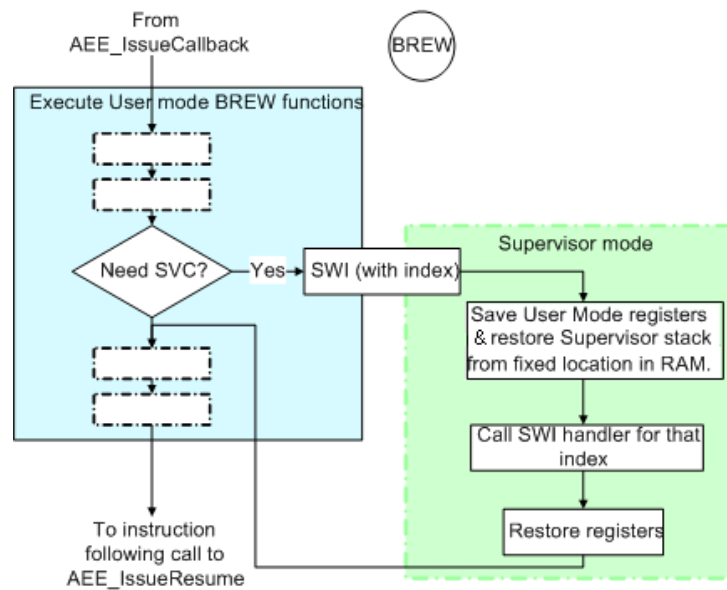
Supervisor-to-User mode context switch



User-to-Supervisor mode

Changing from User mode to Supervisor mode is potentially unsafe and definitely more involved. The safety issues have been discussed in previous sections. The actual mechanism for this transition depends on the hardware, but almost always involves a software interrupt (SWI) whose handler is installed and executed in Supervisor mode. This may be referred to as a "system call", and is illustrated in the following figure

.User-to-Supervisor mode context switch



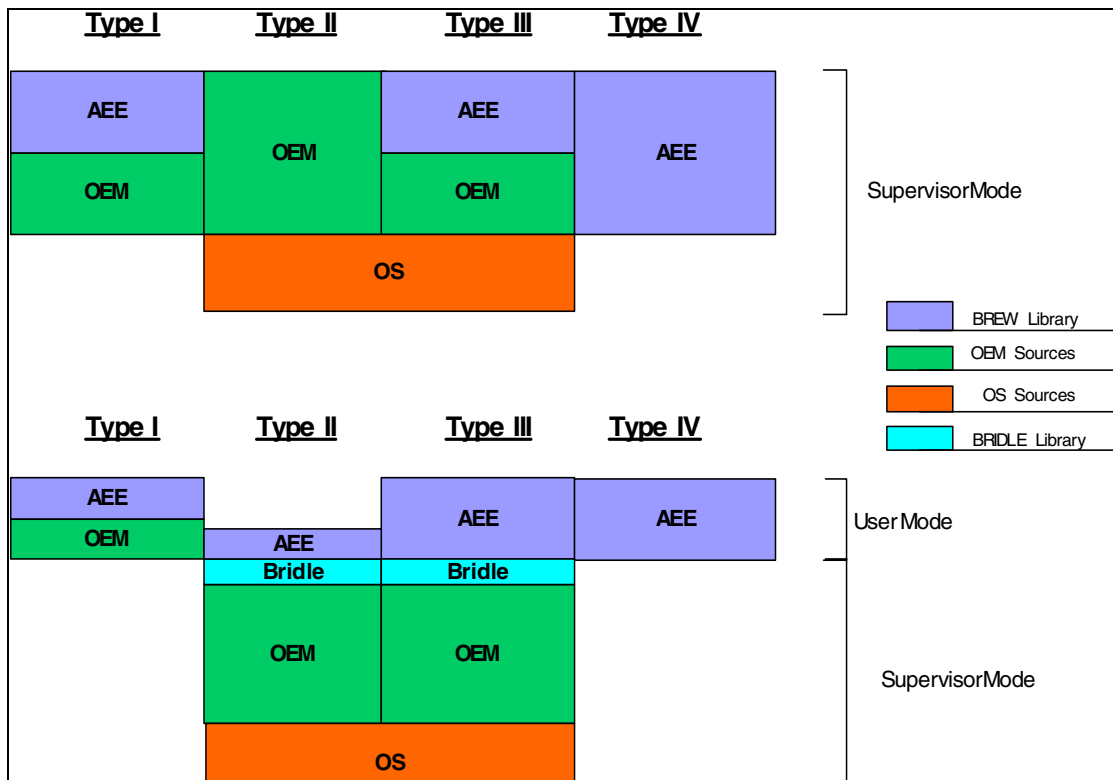
Boundaries

BRIDLE has introduced a new boundary to separate the portions of BREW running in User mode from those running in Supervisor mode. Where this boundary is made depends on individual interfaces, and this section attempts to aid BREW interface developers in establishing this boundary in their interfaces. The interfaces can essentially be categorized into 4 major types, as follows:

- Type I: BREW interfaces that have AEE and OEM layers, but the OEM layers do not require system services (e.g. AEEDB/OEMDB). These interfaces would not require BRIDLE.
- Type II: BREW interfaces that are entirely implemented in OEM (e.g. OEMPosDet) and do access the system. The best way to convert these interfaces to be BRIDLE compliant would be to separate the functions into device independent BREW interface functions (e.g. AEEPosDet) and the device dependent part that requires system services.
- Type III: AEE libraries that access the system via OEM interfaces (e.g. AEENet/OEMSock). For these, the protection may be applied at the AEE-OEM boundary.
- Type IV: Interfaces that are implemented as pure AEE layers and do not access the system (e.g. OEMHeap). These interfaces need not be BRIDLEd and can run entirely in User mode.

The following figure illustrates how BRIDLEization would affect the 4 types of BREW interfaces described above. The diagram on the top illustrates the 4 types of interfaces as they exist today, and the diagram below represents the interfaces after BRIDLE.

Four types of interfaces, before and after BRIDLE



Type II Interfaces

For Type II interfaces, there will be two variations on creating the new AEE layer. In both cases, the end result is a thin OEM layer.

Thin AEE

In the case where the existing OEM layer is relatively thin (doing not much more than calling OS functions), the new AEE layer can be extremely thin. For example, it could be a simple header file that aliases the new AEE function to be the packbridle OEM function, as shown below:

```
#define AEEFoo_Openpackbridle_OEMFoo_Open
```

Then code that used to call OEMFoo_Open() can be changed to simply call AEEFoo_Open().

Thick AEE

In the case where the OEM layer is relatively complex, e.g. implementing a BREW API, the strategy will be a bit different. In this case, the philosophy is to move as much code to user space as possible.

One way to do this would be to copy the contents of the OEM files to the new AEE files, and then create a thin OEM layer (without vtables) that does not much more than call OS functions.

OEM API Design

Whether you are creating a new OEM API (for example, Type II), or BRIDLEing an existing Type III interface, there are some BRIDLE-specific considerations that may impact the OEM API definition:

- Do not pass structs as arguments. The SWI mechanism assumes that all arguments can fit in a 32-bit word, so pass a pointer to struct instead.
- Do not allocate memory in Supervisor mode and return it to User mode, because User mode will not be able to access the memory. Instead, allocate the memory in User mode and pass the pointer down into Supervisor mode. When in supervisor mode, use the `BRIDLE_COPY_*_USER` macros to access the user memory.

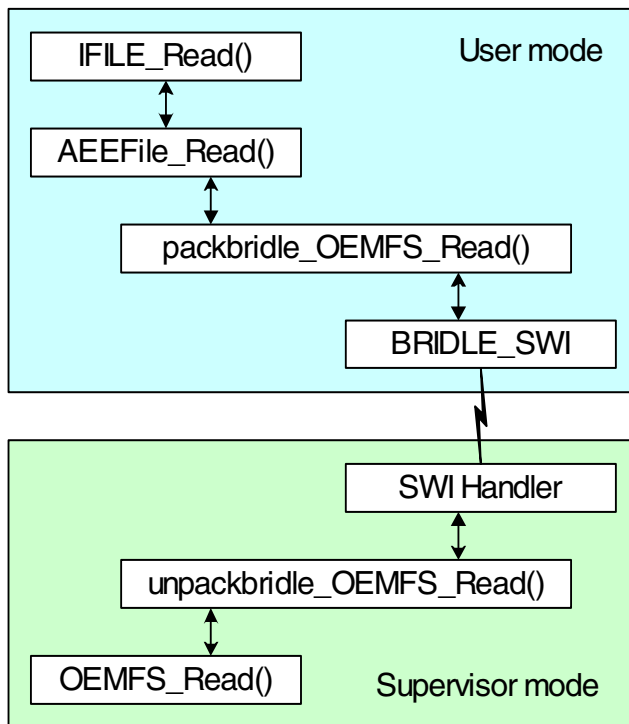
BRIDLE_SWI

BRIDLE_SWI is the function used to perform a User to Supervisor mode context switch. The specific implementation is platform dependent, but it is typically a combination of an assembly routine that invokes a software interrupt, and the corresponding interrupt handler (SWI Handler).

For Arm7 and ARM9, BRIDLE_SWI utilizes registers to pass arguments, but it only has four registers for this purpose. Therefore, any system call that requires more than four arguments needs to be rewritten to only use four arguments. This is achieved by packing arguments four and later into a struct, and passing the address of the struct as the fourth argument.

As an example, consider IFILE_Read(), which is part of a Type III interface. Where AEEFile_Read() used to directly call OEMFS_Read(), there now needs to be a BRIDLE boundary inserted. AEEFile_Read() will call packbridle_OEMFS_Read(), which will pack the arguments and traverse the boundary via BRIDLE_SWI. On the other side, the SWI Handler will pass the registers to unpackbridle_OEMFS_Read(), which will unpack and validate the arguments before calling OEMFS_Read().

Using BRIDLE_SWI to traverse the BRIDLE boundary



BRIDLE Macros

The BRIDLE macros are defined in AEEBridle.h. They can be used to facilitate making the context switch between user and supervisor modes. In practice, you will never need to directly invoke BRIDLE_SWI.

BRIDLEX_*

```
//
// BRIDLEX_* macros - generates both PACKBRIDLE and UNPACKBRIDLE inlines
//
// BRIDLEX_RET(rt,f,t1,v1,n1,...,tn,vn,nn,i,n)
//   X - number of paramters to deal with (e.g. 4)
//   rt - return type (e.g. int32)
//   f - system function (e.g. OEMSocket_Open)
//   t1 - type of first arg (e.g. byte)
//   v1 - first arg verification function (e.g. BRIDLE_CHECK_NONE)
//   n1 - first number of bytes to verify (e.g. 0)
//   ...
//   tn - type of nth arg (e.g. uint16*)
//   vn - nth arg verification function (e.g. BRIDLE_CHECK_WRITEACCESS)
//   nn - nth number of bytes to verify (e.g. sizeof(uint16))
//   i - swi number (e.g. bridle_sock_swi)
//   n - swi index (e.g. BRIDLE_SOCK_SOCKET)
//
// BRIDLEX_VOID(f,t1,...,tn,i,n)
//   X - number of paramters to deal with (e.g. 4)
//   f - system function (e.g. OEMFS_RegRmtAccessChk)
//   t1 - type of first arg (e.g. byte)
//   v1 - first arg verification function (e.g. BRIDLE_CHECK_NONE)
//   n1 - first number of bytes to verify (e.g. 0)
//   ...
//   tn - type of nth arg (e.g. uint16*)
//   vn - nth arg verification function (e.g. BRIDLE_CHECK_WRITEACCESS)
//   nn - nth number of bytes to verify (e.g. sizeof(uint16))
//   i - swi number (e.g. bridle_sock_swi)
//   n - swi index (e.g. BRIDLE_SOCK_FORCE_DORMANCY)
//
#define BRIDLE0_RET(rt,f,i,n) \
PACKBRIDLE0_RET(rt,f,i,n) \
UNPACKBRIDLE0_RET(rt,f,i,n)

#define BRIDLE0_VOID(f,i,n) \
PACKBRIDLE0_VOID(f,i,n) \
```

```
UNPACKBRIDLE0_VOID(f,i,n)

#define BRIDLE1_RET(rt,f,t1,v1,n1,i,n) \
PACKBRIDLE1_RET(rt,f,t1,i,n) \
UNPACKBRIDLE1_RET(rt,f,t1,v1,n1,i,n)

...

#define
BRIDLE13_RET(rt,f,t1,v1,n1,t2,v2,n2,t3,v3,n3,t4,v4,n4,t5,v5,n5,t6,v6,n6,t7,
v7,n7,t8,v8,n8,t9,v9,n9,t10,v10,n10,t11,v11,n11,t12,v12,n12,t13,v13,n13,i,n
) \
PACKBRIDLE13_RET(rt,f,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,i,n) \

UNPACKBRIDLE13_RET(rt,f,t1,v1,n1,t2,v2,n2,t3,v3,n3,t4,v4,n4,t5,v5,n5,t6,v6,
n6,t7,v7,n7,t8,v8,n8,t9,v9,n9,t10,v10,n10,t11,v11,n11,t12,v12,n12,t13,v13,n
13,i,n)
```

PACKBRIDLEX_*

```
//
// PACKBRIDLEX_* macros - generates an inline for the user mode invocation
// of the SWI
//
// PACKBRIDLEX_RET(rt,f,t1,...,tn,i,n)
//   X - number of paramters to deal with (e.g. 4)
//   rt - return type (e.g. int32)
//   f - system function (e.g. OEMSocket_Open)
//   t1 - type of first arg (e.g. byte)
//   ...
//   tn - type of nth arg (e.g. uint16*)
//   i - swi number (e.g. bridle_sock_swi)
//   n - swi index (e.g. BRIDLE_SOCK_SOCKET)
//
// PACKBRIDLEX_VOID(f,t1,...,tn,i,n)
//   X - number of paramters to deal with (e.g. 4)
//   f - system function (e.g. OEMFS_RegRmtAccessChk)
//   t1 - type of first arg (e.g. byte)
//   ...
//   tn - type of nth arg (e.g. uint16*)
//   i - swi number (e.g. bridle_sock_swi)
//   n - swi index (e.g. BRIDLE_SOCK_FORCE_DORMANCY)
//
#define PACKBRIDLE0_RET(rt,f,i,n) \
static __inline rt packbridle_##f(void) \
{ \
    return (rt)BRIDLE_SWI(0,0,0,0,BRIDLE_MAKE_SWI(i,n)); \
}
#define PACKBRIDLE0_VOID(rt,f,i,n) \
static __inline void packbridle_##f(void) \
{ \
```

```

        (void)BRIDLE_SWI(0,0,0,0,BRIDLE_MAKE_SWI(i,n)); \
    }
#define PACKBRIDLE1_RET(rt,f,t1,i,n) \
static __inline rt packbridle_##f(t1 a1) \
{ \
    return (rt)BRIDLE_SWI((uint32)a1,0,0,0,BRIDLE_MAKE_SWI(i,n)); \
}

...

#define
PACKBRIDLE13_RET(rt,f,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,i,n) \
static __inline rt packbridle_##f(t1 a1,t2 a2,t3 a3,t4 a4,t5 a5,t6 a6,t7
a7,t8 a8,t9 a9,t10 a10,t11 a11,t12 a12,t13 a13) \
{ \
    return
(rt)packbridle(BRIDLE_MAKE_SWI(i,n), (uint32)a1, (uint32)a2, (uint32)a3,10, (ui
nt32)a4, (uint32)a5, (uint32)a6, (uint32)a7, (uint32)a8, (uint32)a9, (uint32)a10,
(uint32)a11, (uint32)a12, (uint32)a13); \
}

```

packbridle() inline

PACKBRIDLE4_*() and above use a helper inline to pack the extra parameters into one, since only four arguments are passed through the SWI. There should be no need to call this directly.

```

// useful for packing more than four parameters into four parameters to
traverse the SWI
static __inline uint32 packbridle(uint32 swi, uint32 a1, uint32 a2, uint32
a3, int x, ...)
{
    va_list ap;
    int i;
    uint32 pu[10];

    va_start(ap, x);
    for (i = 0; i < x; ++i) {
        pu[i] = va_arg(ap, uint32);
    }
    va_end(ap);

    return BRIDLE_SWI(a1, a2, a3, (uint32)pu, swi);
}

```

UNPACKBRIDLEX_*

```
//
// UNPACKBRIDLEX_* macros - generates an inline for the SWI to invoke
// Note that there are very few cases in which these would need to
// be used by themselves - instead use the BRIDLEX_* macros below.
//
// UNPACKBRIDLEX_RET(rt,f,t1,v1,n1,...,tn,vn,nn,i,n)
// X - number of paramters to deal with (e.g. 4)
// rt - return type (e.g. int32)
// f - system function (e.g. OEMSocket_Open)
// t1 - type of first arg (e.g. byte)
// v1 - first arg verification function (e.g. BRIDLE_CHECK_NOACCESS)
// n1 - first number of bytes to verify (e.g. 0)
// ...
// tn - type of nth arg (e.g. uint16*)
// vn - nth arg verification function (e.g. BRIDLE_CHECK_WRITEACCESS)
// nn - nth number of bytes to verify (e.g. sizeof(uint16))
// i - swi number (e.g. bridle_sock_swi)
// n - swi index (e.g. BRIDLE_SOCK_SOCKET)
//
// UNPACKBRIDLEX_VOID(f,t1,...,tn,i,n)
// X - number of paramters to deal with (e.g. 4)
// f - system function (e.g. OEMFS_RegRmtAccessChk)
// t1 - type of first arg (e.g. byte)
// v1 - first arg verification function (e.g. BRIDLE_CHECK_NOACCESS)
// n1 - first number of bytes to verify (e.g. 0)
// ...
// tn - type of nth arg (e.g. uint16*)
// vn - nth arg verification function (e.g. BRIDLE_CHECK_WRITEACCESS)
// nn - nth number of bytes to verify (e.g. sizeof(uint16))
// i - swi number (e.g. bridle_sock_swi)
// n - swi index (e.g. BRIDLE_SOCK_FORCE_DORMANCY)
//
#define UNPACKBRIDLE0_RET(rt,f,i,n) \
static __inline uint32 unpackbridle_##f(uint32 swi,uint32 a1,uint32 \
a2,uint32 a3,uint32 a4) \
{ \
    return (uint32)f(); \
}
#define UNPACKBRIDLE0_VOID(f,i,n) \
static __inline uint32 unpackbridle_##f(uint32 swi,uint32 a1,uint32 \
a2,uint32 a3,uint32 a4) \
{ \
    f(); \
    return 0; \
}
#define UNPACKBRIDLE1_RET(rt,f,t1,v1,n1,i,n) \
static __inline uint32 unpackbridle_##f(uint32 swi,uint32 a1,uint32 \
a2,uint32 a3,uint32 a4) \
{ \
    v1(swi,(t1)UNPACKBRIDLE_PARAM_1,n1); \
    return (uint32)f((t1)UNPACKBRIDLE_PARAM_1); \
}
```

```

...

#define
UNPACKBRIDLE13_RET(rt, f, t1, v1, n1, t2, v2, n2, t3, v3, n3, t4, v4, n4, t5, v5, n5, t6, v6,
n6, t7, v7, n7, t8, v8, n8, t9, v9, n9, t10, v10, n10, t11, v11, n11, t12, v12, n12, t13, v13, n
13, i, n) \
static __inline uint32 unpackbridle_##f(uint32 swi, uint32 a1, uint32
a2, uint32 a3, uint32 a4) \
{ \
    v1(swi, (t1)UNPACKBRIDLE_PARAM_1, n1); \
    v2(swi, (t2)UNPACKBRIDLE_PARAM_2, n2); \
    v3(swi, (t3)UNPACKBRIDLE_PARAM_3, n3); \
    v4(swi, (t4)UNPACKBRIDLE_PARAM_4, n4); \
    v5(swi, (t5)UNPACKBRIDLE_PARAM_5, n5); \
    v6(swi, (t6)UNPACKBRIDLE_PARAM_6, n6); \
    v7(swi, (t7)UNPACKBRIDLE_PARAM_7, n7); \
    v8(swi, (t8)UNPACKBRIDLE_PARAM_8, n8); \
    v9(swi, (t9)UNPACKBRIDLE_PARAM_9, n9); \
    v10(swi, (t10)UNPACKBRIDLE_PARAM_10, n10); \
    v11(swi, (t11)UNPACKBRIDLE_PARAM_11, n11); \
    v12(swi, (t12)UNPACKBRIDLE_PARAM_12, n12); \
    v13(swi, (t13)UNPACKBRIDLE_PARAM_13, n13); \
    return (uint32) f((t1)UNPACKBRIDLE_PARAM_1, \
        (t2)UNPACKBRIDLE_PARAM_2, \
        (t3)UNPACKBRIDLE_PARAM_3, \
        (t4)UNPACKBRIDLE_PARAM_4, \
        (t5)UNPACKBRIDLE_PARAM_5, \
        (t6)UNPACKBRIDLE_PARAM_6, \
        (t7)UNPACKBRIDLE_PARAM_7, \
        (t8)UNPACKBRIDLE_PARAM_8, \
        (t9)UNPACKBRIDLE_PARAM_9, \
        (t10)UNPACKBRIDLE_PARAM_10, \
        \
        (t11)UNPACKBRIDLE_PARAM_11, \
        (t12)UNPACKBRIDLE_PARAM_12, \
        (t13)UNPACKBRIDLE_PARAM_13); \
}

```

UNPACKBRIDLE_PARAM_X

```

//
// UNPACKBRIDLE_PARAM_X macros - used for accessing packed parameters
// (also useful for validating a pointer whose length is specified by an-
other parameter)
#define UNPACKBRIDLE_PARAM_1 (a1)
#define UNPACKBRIDLE_PARAM_2 (a2)
#define UNPACKBRIDLE_PARAM_3 (a3)
#define UNPACKBRIDLE_PARAM_4 (((uint32*)a4)[0])
#define UNPACKBRIDLE_PARAM_5 (((uint32*)a4)[1])
#define UNPACKBRIDLE_PARAM_6 (((uint32*)a4)[2])
#define UNPACKBRIDLE_PARAM_7 (((uint32*)a4)[3])
#define UNPACKBRIDLE_PARAM_8 (((uint32*)a4)[4])
#define UNPACKBRIDLE_PARAM_9 (((uint32*)a4)[5])

```

```
#define UNPACKBRIDLE_PARAM_10 (((uint32*)a4)[6])
#define UNPACKBRIDLE_PARAM_11 (((uint32*)a4)[7])
#define UNPACKBRIDLE_PARAM_12 (((uint32*)a4)[8])
#define UNPACKBRIDLE_PARAM_13 (((uint32*)a4)[9])
```

BRIDLE_CHECK_*

These macros are used in `unpackbridle_*`() to validate memory (i.e. that the memory range falls entirely within user space). Note that they do not return in the case of a failure. The `swi` parameter is used to disable checking in the case that the BRIDLE call was entered from supervisor mode.

- Verify that the pointer can not be accessed in user mode (not too useful right now):

```
BRIDLE_CHECK_NOACCESS(swi, p, len)
```

- Verify that the pointer can be read in user mode:

```
BRIDLE_CHECK_READACCESS(swi, p, len)
```

- Verify that the pointer can be written in user mode:

```
BRIDLE_CHECK_WRITEACCESS(swi, p, len)
```

- Verify (only if not-null) that the pointer (non-zero) can be read in user mode:

```
BRIDLE_CHECK_READACCESS_NZ(swi, p, len)
```

- Verify (only if non-null) that the pointer (non-zero) can be written in user mode:

```
BRIDLE_CHECK_WRITEACCESS_NZ(swi, p, len)
```

- Do not verify (e.g. pointer not used, not a pointer):

```
BRIDLE_CHECK_NONE(swi, p, len)
```

- Verify an IOCTL from AEEIOCTL.h:

```
BRIDLE_CHECK_IOCTL(swi, p, len)
```

- Verify a string for read access (uses safe strlen):

```
BRIDLE_CHECK_READ_STRING_ACCESS(swi, p, unused)
```

- Verify an exact value (e.g. callback function pointer):

```
BRIDLE_CHECK_VALUE(swi, v, V)
```

- Verify that an AEEHandle (h) refers to a valid entry in the AEEInstanceList (pIL), and if so converts it to the corresponding OEMINSTANCE:

```
BRIDLE_CHECK_HANDLE(swi, h, pIL)
```

- Verify that an hAEEkCB (kcb) refers to a valid handle, and if so converts it to the corresponding AEECallback*:

```
BRIDLE_CHECK_AEEKCB(swi, kcb, unused)
```

See the examples for usage.

BRIDLE_COPY_*_USER

These macros are used to copy memory to and from user space; for example, a callback from supervisor mode that needs to copy data to a user mode buffer. For now, the copies are direct, but they will involve virtual memory translations in Bridle II.

- Verify the user mode pointer (to) for write and copy from supervisor mode:

```
BRIDLE_COPY_TO_USER(to, from, len)
```

- Verify the user mode pointer (from) for read and copy to supervisor mode:

```
BRIDLE_COPY_FROM_USER(to, from, len)
```

- If the pointers have already been validated, there are versions that do not verify:

```
COPY_TO_USER(to, from, len)
```

- COPY_FROM_USER(to, from, len)

BRIDLE_SUBSYS_*

These macros provide facilities to build a typical BRIDLE subsystem. The key facility of BRIDLE_SUBSYS_* macros is a naming convention that can help eliminate errors caused by copy-and-paste code editing. BRIDLE has no type checking for the SWI handlers of each subsystem, so things like initialization of a subsystem's SWI handler vector table is error-prone. Since the subsystem's SWI number and the subsystem's initialization function are in the global namespace, the BRIDLE_SUBSYS_* macros provide a naming convention that can keep the names from colliding.

Among the conventions enforced are:

- The subsystem's SWI number is named as `g_bridle_<subsystem>_subsys_no`

- Array indices used for accessing the subsystem's SWI handler table are named `BRIDLE_<subsystem>_<function>`
- The size of the subsystem's SWI handler table is `BRIDLE_<subsystem>_COUNT`

With these conventions in place, writing a typical BRIDLE subsystem is a matter of a few declarations, as shown below. See Examples for more information.

```
// construct a constant name that's subsystem+function specific, to be stuck
// in an enum { };, used for initializing the subsystem's SWI handler table,
// and naming sys calls
#define BRIDLE_SUBSYS_HANDLER_IDX(name, func) BRIDLE_##name##_##func

// declares a subsystem-specific number name
#define BRIDLE_SUBSYS_NO(name) g_bridle_##name##_subsys_no

// names the subsystem's handler table
#define BRIDLE_SUBSYS_HANDLER_TABLE(name) g_##name##_handlers

// declares name the subsystem's SWI handler table
#define BRIDLE_DECLARE_SUBSYS_HANDLER_TABLE(name) \
    static PFNSWIHANDLER
BRIDLE_SUBSYS_HANDLER_TABLE(name) [BRIDLE_##name##_COUNT]

// the default implementation of the subsystem's top-level SWI handler would
// look like this
#define BRIDLE_DECLARE_GENERIC_SUBSYS_SWIHANDLER(name) \
static uint32 bridler_##name##_swihandler(uint32 swi, uint32 a1, uint32 a2,
uint32 a3, uint32 a4)\
{\
    return bridler_generic_swihandler(swi, a1, a2, a3, a4, \
ARRAY_SIZE(BRIDLE_SUBSYS_HANDLER_TABLE(name)), \
    BRIDLE_SUBSYS_HANDLER_TABLE(name));\
}

// the call back to bridler to register your subsystem that uses BRIDLE_SUBSYS_
// naming
#define BRIDLE_REGISTER_SUBSYS(name) \
    bridler_RegisterSubSystem(bridler_##name##_swihandler,
&BRIDLE_SUBSYS_NO(name))

// initializes one entry in the subsystems SWI handler table
#define BRIDLE_INIT_SUBSYS_HANDLER_TABLE(name, func) \
    BRIDLE_SUBSYS_HANDLER_TABLE(name) [BRIDLE_SUBSYS_HANDLER_IDX(name, func)] =
unpackbridler_##func;
```

OEMFoo.h

To start this example, assume that OEMFoo provides some simple methods. More complex methods will be added later in the example.

```
extern OEMINSTANCE OEMFoo_Open(const char name[]);
extern int OEMFoo_Read(OEMINSTANCE f, char buf[], int size);
extern int OEMFoo_Write(OEMINSTANCE f, const char buf[], int size);
extern void OEMFoo_Close(OEMINSTANCE f);
```

BRIDLEInit.h (or BRIDLEInit_OEM.h)

Add a new prototype:

```
extern int    bridle_foo_RegisterSubSystem (void *po);
```

Note that some device builds (e.g. 6050/6100) may have their own copy of BRIDLEInit_OEM.h that should be modified as needed.

BRIDLEInit.c (or BRIDLEInit_OEM.c)

Bridle initialization takes place before AEEInit(). In order to register the module SWI handlers, add an entry to gBridleInit[] (or gOEMBridleInit[]), which will be iterated over during Bridle initialization. Use feature macros if appropriate.

```
...
static const PFNBIDLEINIT gBridleInit[] = {
...
#ifdef FEATURE_BREW_FOO
    bridle_foo_RegisterSubSystem,
#endif // FEATURE_BREW_FOO
...
};
```

Note that some device builds (e.g. 6050/6100) may have their own copy of BRIDLEInit_OEM.c that should be modified as needed.

bridle_foo.c

At a minimum, the unpack file will need to initialize the module SWI handlers. Since this is a new file, it will need to be added to the appropriate part of the build system (e.g. aeebridle.min).

```
#include iBRIDLEInit.h
#include ibridle_foo.h
```

The SWI number needs to be user read-only, which is achieved via a pragma that tells the linker to use a different rule in the scatter load file:

```
#if __ARMCC_VERSION >= 120000 // ADS 1.2
#pragma arm section zidata = "bridle_svc_zi_usr_ro"
#endif // __ARMCC_VERSION >= 120000 // ADS 1.2
uint32 BRIDLE_SUBSYS_NO(foo);
#if __ARMCC_VERSION >= 120000 // ADS 1.2
#pragma arm section zidata
#endif // __ARMCC_VERSION >= 120000 // ADS 1.2

// declare/define the table of function pointers
BRIDLE_DECLARE_SUBSYS_HANDLER_TABLE(foo);

// this module employs the generic swi handling logic
BRIDLE_DECLARE_GENERIC_SUBSYS_SWIHANDLER(foo)

int bridle_foo_RegisterSubSystem(void *po)
{
    BRIDLE_INIT_SUBSYS_HANDLER_TABLE(foo, OEMFoo_Open);
    BRIDLE_INIT_SUBSYS_HANDLER_TABLE(foo, OEMFoo_Read);
    BRIDLE_INIT_SUBSYS_HANDLER_TABLE(foo, OEMFoo_Write);
    BRIDLE_INIT_SUBSYS_HANDLER_TABLE(foo, OEMFoo_Close);

    return BRIDLE_REGISTER_SUBSYS(foo);
}
```

bridle_foo.h

```
#ifndef _BRIDLE_FOO_H_
#define _BRIDLE_FOO_H_

#include "OEMFoo.h"
#include "AEEBridle.h"
```

Enumerate the methods for the module SWI handler:

```
enum {
    BRIDLE_SUBSYS_HANDLER_IDX(foo, OEMFoo_Open),
    BRIDLE_SUBSYS_HANDLER_IDX(foo, OEMFoo_Read),
    BRIDLE_SUBSYS_HANDLER_IDX(foo, OEMFoo_Write),
    BRIDLE_SUBSYS_HANDLER_IDX(foo, OEMFoo_Close),
    BRIDLE_foo_COUNT
};

extern uint32 RIDLE_SUBSYS_NO(foo);
```

Declare the `packbridle_*` and `unpackbridle_*` inline functions via the BRIDLE macros:

```
BRIDLE_SUBSYS_1_RET(foo, OEMINSTANCE, OEMFoo_Open,
    const char*, BRIDLE_CHECK_READ_STRING_ACCESS, 0)

BRIDLE_SUBSYS_3_RET(foo, int, OEMFoo_Read,
    OEMINSTANCE, BRIDLE_CHECK_NONE, 0,
    char*, BRIDLE_CHECK_WRITEACCESS, UNPACKBRIDLE_PARAM_3,
    int, BRIDLE_CHECK_NONE, 0)

BRIDLE_SUBSYS_3_RET(foo, int, OEMFoo_Write,
    OEMINSTANCE, BRIDLE_CHECK_NONE, 0,
    const char*, BRIDLE_CHECK_READACCESS, UNPACKBRIDLE_PARAM_3,
    int, BRIDLE_CHECK_NONE, 0)

BRIDLE_SUBSYS_1_VOID(foo, OEMFoo_Close,
    OEMINSTANCE, BRIDLE_CHECK_NONE, 0)

#endif // _BRIDLE_FOO_H_
```

AEEFoo.c

```
#include ibridle_foo.h
```

Replace prior `OEMFoo_*` calls with the `packbridle_*` variants. These will instantiate the corresponding inline functions from `bridle_foo.h`.

```
...
    f = packbridle_OEMFoo_Open(ifoobarî);
...
    n = packbridle_OEMFoo_Read(f, buf, sizeof(buf));
...
    m = packbridle_OEMFoo_Write(f, buf, n);
...
    packbridle_OEMFoo_Close(f);
```

...

Complex parameter checking

Consider the following function which takes an array of structures that in turn contain pointers that need to be validated:

```
typedef struct {
    char* buf;
    int size;
} IOVec;

extern int OEMFoo_ReadV(OEMINSTANCE f, IOVec bufs[], int count);
```

The BRIDLE_CHECK_*ACCESS macros are limited to checking a single pointer and length. In this case, declare only the PACKBRIDLE macro in bridle_foo.h:

```
PACKBRIDLE_SUBSYS_3_RET(foo, int, OEMFoo_ReadV,
    OEMINSTANCE f,
    IOVec*,int)
```

The unpackbridle function will need to be written by hand in unpackbridle_foo.c:

```
static uint32 unpackbridle_OEMFoo_ReadV(uint32 swi, uint32 a1, uint32 a2,
uint32 a3, uint32 a4)
{
    OEMINSTANCE f = (OEMINSTANCE)UNPACKBRIDLE_PARAM_1;
    IOVec* bufs = (IOVec*)UNPACKBRIDLE_PARAM_2;
    int count = (int)UNPACKBRIDLE_PARAM_3;
    int i;

    BRIDLE_CHECK_READACCESS(swi, bufs, count*sizeof(IOVec));
    for (i = 0; i < count; ++i) {
        BRIDLE_CHECK_WRITEACCESS(swi, bufs[i].buf, bufs[i].size);
    }

    return (uint32)OEMFoo_ReadV(f, bufs, count);
}
```

Callbacks

Static Callbacks

Since OEMFoo.c will be operating in SVC mode, is it OK to utilize a static callback in AEEFoo.c? It is safe, if certain guidelines are followed:

- The callback function pointer needs to be validated, such that arbitrary user code is not executed in system mode. This usually involves converting a static function to extern, such that the symbol will be available to OEMFoo.c. The callback itself cannot be dynamic.
- The callback function can copy data from supervisor mode to user mode. Additionally, if the data sink is dynamically allocated, it must be validated before the copy.
- The callback may post a resume (which will occur in user mode).
- No other operations may take place in the callback.

As an example, suppose that AEEFoo registers a callback with OEMFoo:

```
typedef struct {
    char* buf;
    int size;
} FooEvent;

typedef void (PFNFOO*)(void* pUser, const FooEvent* pEvent);

extern void OEMFoo_Register(OEMINSTANCE f, PFNFOO cb, void* pUser);
```

This function would be BRIDLED as follows:

```
BRIDLE_SUBSYS_3_VOID(foo, OEMFoo_Register,
    OEMINSTANCE f, BRIDLE_CHECK_NONE, 0,
    PFNFOO, BRIDLE_CHECK_VALUE, AEEFoo_EventCB,
    void*, BRIDLE_CHECK_READACCESS, sizeof(AEEFoo))
```

The callback would be registered like this:

```
CALLBACK_Init(&me->cbEvent, AEEFoo_HandleEvent, me);
backbridle_OEMFoo_Register(me->f, AEEFoo_EventCB, me);
```

The callback should look something like this:

```
void AEEFoo_EventCB(void* pUser, const FooEvent* pEvent)
{
    AEEFoo* me = (AEEFoo*)pUser;

    BRIDLE_COPY_TO_USER(me->buf, pEvent->buf, pEvent->size);
    AEE_ResumeCallback(&me->cbEvent, 0);
}
```

AEECallbacks

Is it safe to pass an AEECallback* across the BRIDLE boundary? No, this is unsafe since the contents of the AEECallback (e.g. pNext) may be modified at any time by user code.

Since the AEECallback mechanism is a convenient one, a facility has been developed to make use of them across a BRIDLE boundary: AEEkCB.

Let's consider a readable method:

```
extern void OEMFoo_Readable(OEMINSTANCE f, AEECallback* pcb);
```

On the USR side, change the AEECallback member of the instance data to hAEEkCB, and change CALLBACK_Init() to AEEkCB_Create(). Since this handle now refers to a dynamically allocated (SVC) AEECallback, be sure to AEEkCB_Delete() it when finished:

```
struct {
    ...
    hAEEkCB kcb;
    ...
} AEEFoo;
...
static void AEEFoo_ReadableCB(void* pv);

int AEEFoo_Init(void)
{
    int nErr;

    nErr = AEEkCB_Create(&me->kcb, AEEFoo_ReadableCB, me);
    if (SUCCESS != nErr) {
        return nErr;
    }
    ...
}

int AEEFoo_Delete(void)
{
    ...
    AEEkCB_Delete(&me->kcb);
}
```

```
}

```

In `bridle_foo.h`, the new macro will look like this:

```
BRIDLE_SUBSYS_2_VOID(foo, OEMFoo_Readable,
    OEMINSTANCE, BRIDLE_CHECK_NONE, 0,
    AEECallback*, BRIDLE_CHECK_AEEKCB, 4)

```

In `AEEFoo.c`, it is called by passing `hAEEkCB` by value, instead of the `AEECallback` by reference:

```
packbridle_OEMFoo_Readable(me->f, me->kcb);

```

`BRIDLE_CHECK_AEEKCB` will first validate the handle, and then call `OEMFoo_Readable()` with the underlying SVC `AEECallback`, which `OEMFoo.c` can use just like before, with the exception that the resume now needs to be done in supervisor mode:

```
AEE_SVC_RESUME(pcb, 0);

```

Handles

So far it has been assumed that the `OEMINSTANCE` returned from `OEMFoo_Open()` is a small integer which is validated by the `OEMFoo_*`() calls. What if it is instead a pointer from the system heap? This is OK from an access point of view, as `AEEFoo.c` never tries to dereference the pointer, but how does `OEMFoo_*`() ensure that the pointer is one that was returned from `OEMFoo_Open()`?

The solution is that the `OEMINSTANCE` would be converted to a small integer handle in `unpackbridle_OEMFoo_Open()`, and from a small integer handle to the `OEMINSTANCE` in the other `unpackbridle_OEMFoo_*`() functions.

In this case, `bridle_foo.h` would be changed such that the macro for `OEMFoo_Open` and `OEMFoo_Close` would change from a `BRIDLE1_RET` to `PACKBRIDLE1_RET`, and the validation of the first parameter in the other macros would change from `BRIDLE_CHECK_NONE` to `BRIDLE_CHECK_HANDLE`:

```
extern AEEInstanceList gFooInstanceList;

PACKBRIDLE_SUBSYS_1_RET(foo, OEMINSTANCE, OEMFoo_Open,
    const char*)

BRIDLE_SUBSYS_3_RET(foo, int, OEMFoo_Read,

```

```
OEMINSTANCE, BRIDLE_CHECK_HANDLE, &gFooInstanceList,
char*, BRIDLE_CHECK_WRITEACCESS, UNPACKBRIDLE_PARAM_3,
int, BRIDLE_CHECK_NONE, 0)

PACKBRIDLE_SUBSYS_1_VOID(foo, OEMFoo_Close,
                        OEMINSTANCE)

...
```

In `bridle_foo.c`, the instance list would be declared and `unpackbridle_OEMFoo_Open()` and `unpackbridle_OEMFoo_Close()` would be written as follows:

```
static AEEInstance gpInstances[16];
AEEInstanceList gFooInstanceList = { gpInstances, sizeof(gpInstances) /
sizeof(gpInstances[0]), 0 };

static uint32 unpackbridle_OEMFoo_Open(uint32 swi, uint32 a1, uint32 a2,
uint32 a3, uint32 a4)
{
    const char* szName = (const char*)UNPACKBRIDLE_PARAM_1;
    OEMINSTANCE f;
    uint32 h;
    boolean bRet;

    BRIDLE_CHECK_READ_STRING_ACCESS(swi, szName, 0);

    // create the instance
    f = OEMFoo_Open(szName);
    if ((OEMINSTANCE)0 == f) {
        return 0;
    }

    // OEMINSTANCE is a pointer from the system heap, which we really
    // don't want to pass to user mode, as they will be hard to validate
    // when they come back to system mode. So, we convert the pointer to
    // an integer "handle" that is easier to validate without any serious
    // performance penalty in the nominal case.
    bRet = AEEHandle_To(&gFooInstanceList, f, &h);
    if (FALSE == bRet) {
        (void)OEMFoo_Close(f);
        return 0;
    }

    return h;
}

static uint32 unpackbridle_OEMFoo_Close(uint32 swi, uint32 a1, uint32 a2,
uint32 a3, uint32 a4)
{
    uint32 h = (uint32)UNPACKBRIDLE_PARAM_1;
    OEMINSTANCE f = (OEMINSTANCE)h;
```

```
BRIDLE_CHECK_HANDLE(swi, f, &gFooInstanceList);

OEMFoo_Close(f);

(void)AEEHandle_Clear(&gFooInstanceList, h);

return 0;
}
```

Scatter Load

Since OEMFoo.c contains supervisor mode data (e.g. `gpInstances`), we need the data to end up in the supervisor portion of the image. It also makes debugging easier if the code is also in the supervisor region, so any data abort will happen at the point of entry, and not further down the stack when data is actually accessed.

Edit the appropriate scatter load file (e.g. `m6100b_ram.scl`), using OEMSock.o as a template. Wherever you find an entry for OEMSock.o, duplicate the line and change `îSockî` to `îFooî` (if you don't see OEMSock.o, your build is not yet BRIDLED). For example, the following line:

```
OEMSock.o (+RW)
```

Will become two lines:

```
OEMSock.o (+RW)
```

```
OEMFoo.o (+RW)
```

There are typically three instances: code (+RO), initialized data (+RW), and uninitialized data (+ZI).

Also, if `bridle_foo.c` is not part of `AEEBridle.lib` (e.g. static extension), entries will need to be created for `bridle_foo.o`, as well.

Other Considerations

Supervisor mode callback mechanisms

Supervisor mode code can not use the standard callback mechanisms for its own uses, as the callback will occur in user mode and a data abort will result.

Supervisor mode variants of timers and resumes have been created, and have exactly the same semantics except that the callback will execute in supervisor mode.

Supervisor mode timers:

```
extern int AEE_SetSvcTimerCallback(int32 nMSecs, AEECallback* pcb);
```

Supervisor mode resumes:

```
extern void AEE_ResumeSvcCallback(AEECallback* pcb);
```

Supervisor mode system callbacks:

```
extern void AEE_RegisterSvcSystemCallback(AEECallback* pcb, int nType);  
extern boolean AEE_IssueSvcSystemCallback(int nType);
```

Locking interrupts

User mode is prohibited from locking interrupts (e.g. INTLOCK()).

If interrupts must truly be locked (e.g. vocoder), then this must be done in supervisor mode, and unlocked before returning to user mode.

Locking interrupts has traditionally been used for making data access thread safe. In these cases, use `AEECriticalSection` (for static data) or `AEEMutex` (for dynamic data) instead. For pure supervisor mode data, `OEMCriticalSection` or `OEMMutex` may be used directly to avoid a bridge transition.

Interrupt service routines

As a result of BRIDLE activities, the BREW dispatcher has been reworked in many ways.

One change was to replace all OEMMutex_Lock() calls with equivalent AEECriticalSection_Enter() calls. In MSM based versions of BREW, OEMMutex_Lock() resulted in INTLOCK(), while AEECriticalSection_Enter() resolves to rex_enter_crit_sect().

While these changes result in better performance of the dispatcher (and BREW in general), rex_enter_crit_sect() is not compatible with ISRs, as there is no task associated with an ISR.

The end result is that ISRs are no longer allowed to call directly into the BREW dispatcher (e.g. AEE_ResumeCallback()). Instead, ISRs need to signal a task to perform this work on their behalf.

While this may appear to introduce additional latency in this case, the BREW dispatcher has better latency characteristics now, so the end result is roughly the same.

Static applications

Historically, static applications have had the luxury of directly making use of the underlying OS. However, with BRIDLE, all callbacks and events are dispatched in user mode, so this is no longer possible.

There are two options for dealing with static apps (e.g. CoreApp) that make OS calls (e.g. CM client):

1. Use an existing BREW interface (e.g. ITAPI), which is already bridled
2. Create a new static extension class and bridle it

Code review

The security related portions of BRIDLE demand extra scrutiny during code review.

If it has been a while, reread section 2 on Responsibility before reviewing the code.

Pay particular attention to parameter validation (`BRIDLE_CHECK_*`()), and the operations performed in supervisor mode callbacks.

OAT Tests

Rigorous OAT tests need to be created/updated for any interface that will ship in source code form or be otherwise exposed to application developers.