Callback Hooks

by David Junod

The *callback* features of Release 2 provide a standard means for applications to extend the functionality of libraries, devices, and their applications. This standard makes it easy for the operating system to use custom modules from different high level programming languages as part of the operating system. For example, the layers library, which takes care of treating a display as a series of layered regions, allows an application to attach a pattern function to a display layer. Instead of filling in the background of a layer with the background color, the layers library calls the custom pattern function which fills in the layer display with a custom background pattern.

Callback Hook Functions and Structures

An application passes a custom function in the form of a *callback Hook* (from *<utility/hooks.h>*):

```
/* Standard hook structure */
struct Hook
{
    struct MinNode h_MinNode;
    ULONG (*h_Entry)();     /* stub function entry point */
    ULONG (*h_SubEntry)();  /* the custom function entry point */
    VOID *h_Data;           /* owner specific */
};
```

h_MinNode - This field is reserved for use by the module that will call the Hook.

h_Entry - This is the address of the Hook *stub*. When the OS calls a callback function, it puts parameters for the callback function in CPU registers A0, A1, and A2. This makes it tough for higher level language programmers to use a callback function because most higher level languages don't have a way to manipulate CPU registers directly. The solution is a stub function which first copies the parameters from the CPU registers to a place where a high level language function can get to them. The stub function then calls the callback function. Typically, the stub pushes the registers onto the stack in a specific order and the high level language callback function pops them off the stack.

h_SubEntry - This is the address of the actual callback function that the application has defined. The stub calls this function.

h_Data - This field is for the application to use. It could point to a global storage structure that the callback function utilizes.

There is only one function defined in *utility.library* for callback functions.

```
ULONG CallHookPkt(struct Hook *hook, VOID *object, VOID *paramPkt);
                               A0           A2             A1
```

This function invokes a standard callback Hook function.

Simple Callback Hook Usage

A Hook function must accept the following three parameters in these specific registers:

A0 Pointer to the Hook structure.
A2 Pointer to an object to manipulate. The object is context specific.
A1 Pointer to a message packet. This is also context specific.

For a callback function written in C, the parameters should appear in this order:

```
myCallbackFunction(Pointer to Hook (A0),
                   Pointer to Object (A2),
                   Pointer to message (A1));
```

This is because the standard C stub pushes the parameters onto the stack in the following order: A1, A2, A0. The following assembly language routine is a callback stub for C:

```
    INCLUDE 'exec/types.i'
    INCLUDE 'utility/hooks.i'

    xdef        _hookEntry

    _hookEntry:
        move.l  a1,-(sp)                ; push message packet pointer
        move.l  a2,-(sp)                ; push object pointer
        move.l  a0,-(sp)                ; push hook pointer
        move.l  h_SubEntry(a0),a0       ; fetch actual Hook entry point ...
        jsr     (a0)                    ; and call it
        lea     12(sp),sp               ; fix stack
        rts
```

If your C compiler supports registerized parameters, your callback functions can get the parameters directly from the CPU registers instead of having to use a stub to push them on the stack. The following C language routine uses registerized parameters to put parameters in the right registers. This routine requires a C compiler that supports registerized parameters.

```
    #include <exec/types.h>
    #include <utility/hooks.h>

    #define     ASM     __asm
    #define     REG(x)  register __ ## x

    /* This function converts register-parameter hook calling
     * convention into standard C conventions.  It requires a C
     * compiler that supports registerized parameters, such as
     * SAS/C 5.xx or greater.
     */
```

```
ULONG ASM hookEntry(REG(a0) struct Hook *h, REG(a2) VOID *o, REG(a1) VOID *msg)
{
    return ((*h->h_SubEntry)(h, o, msg));
}
```

A callback function is executed on the context of the module that invoked it. This usually means that callback functions cannot call functions that need to look at environment specific data. For example, printf() needs to look at the current process's input and output stream. Entities like Intuition have no input and output stream. The limitations on a callback function depend heavily upon the subsystem that is using them. See that subsystem's documentation for more information.

For the callback function to access any of its global data, it needs to make sure the CPU can find the function's data segment. It does this by forcing the function to load the offset for the program's data segment into CPU register A4. See your compiler documentation for details.

The following is a simple function that can be used as a callback Hook.

```
ULONG MyFunction (struct Hook *h, VOID *o, VOID *msg)
{
    /* A SASC and Manx function that obtains access to the global data segment */
    geta4();

    /* Debugging function to send a string to the serial port */
    KPrintF("Inside MyFunction()\n");

    return (1);
}
```

The next step is to initialize the Hook for use. This basically means that the fields of the Hook structure must be filled with appropriate values.

The following simple function initializes a Hook structure.

```
/* This simple function is used to initialize a Hook */
VOID InitHook (struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the hook fields */
        h->h_Entry = (ULONG (*)()) hookEntry;
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}
```

The following is a simple example of initializing and using a callback Hook function.

```
/* hooks1.c
 * Simple hook example
 * Copyright (C) 1991 Commodore-Amiga, Inc.
 *
 */

#include <exec/types.h>
#include <exec/libraries.h>
#include <utility/hooks.h>
#include <clib/exec_protos.h>
```

```
#include <clib/utility_protos.h>

extern struct Library *SysBase;
struct Library *UtilityBase;

#define ASM     __asm
#define REG(x)  register __ ## x

/* This function converts register-parameter Hook calling
 * convention into standard C conventions.  It requires a C
 * compiler that supports registerized parameters, such as
 * SAS/C 5.xx or greater.
 */
ULONG ASM
hookEntry(REG(a0) struct Hook *h, REG(a2) VOID *o, REG(a1) VOID *msg)
{
    return ((*h->h_SubEntry)(h, o, msg));
}

/* This simple function is used to initialize a Hook */
VOID InitHook (struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the Hook fields */
        h->h_Entry = (ULONG (*)()) hookEntry;
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}

/* This function only prints out a message to the serial port indicating that
 * we are inside the callback function.  Note that we cannot use printf() or
 * any other functions that use standard I/O with any of the system modules that
 * support callback Hooks, because there is no guarantee that there would
 * be a valid Output() channel. */

ULONG MyFunction (struct Hook *h, VOID *o, VOID *msg)
{
    /* Obtain access to the global data segment */
    geta4();

    /* Debugging function to send a string to the serial port */
    KPrintF("Inside MyFunction()\n");

    return (1);
}

int main (int argc, char **argv)
{
    struct Hook h;

    /* Open the utility library */
    if (UtilityBase = OpenLibrary ("utility.library", 36))
    {
        /* Initialize the callback Hook */
        InitHook (&h, MyFunction, NULL);

        /* Use the utility library function to invoke the Hook */
        CallHookPkt (&h, NULL, NULL);

        /* Close utility library now that we’re done with it */
        CloseLibrary (UtilityBase);
    }
    else
    {
        /* Display an error message */
        printf ("Couldn’t open utility.library\n");
    }
}
```