Table of Contents

1. Introduction

Applications using libraries directly or indirectly are susceptible to the system forcing a soft reset when the module is removed while the application is running. The soft reset is caused by the system responding to an interrupt generated by the module removal. When the module is removed, the Handspring extension system event handler detects that the module has been removed but the library has not been properly closed by the application. The Handspring extension system event handler forces a soft reset to prevent system instability. This effect of resetting the system during module removal produces an unpleasant user experience due to the long reset time. Therefore applications must protect the system from soft reset by properly closing the library during module removal.

The problem is due to Palm OS lack of library management facilities. Palm OS does not provide any high level logic to track which libraries were opened by which applications, and to automatically close and uninstall libraries that were opened by an application after that application exits. The Handspring extension implements a check-sum verification of the library environment to detect if a library was left open. If open libraries are detected upon Springboard module removed, the system will force a soft reset. Therefore applications and libraries must be designed to track the usage of libraries and properly close them when the module is removed.

2. Application Environments That Are Safe From Soft Reset

When the system tries to access the Springboard memory space while a module is not present, the system will incur a bus fault. This system will examine the its operating environment to determine whether it must force soft reset to avoid a corrupted system environment. There are two common application environments identified below that are safe from bus fault. The two cases below represent implementation of common content-only applications.

2.1. Safe Case 1

An application that executes in handheld memory is safe from bus error fault if (a) the application does not access the module and (b) does not use libraries not from the module. (This is usually the case for many of the commonly available Palm OS applications that do not access the module or any specialized hardware.) Because no bus error fault is generated by module removal, the system does not produce a soft reset. Thus the application and libraries are unaffected. The application and libraries are closed and removed correctly when exiting later.

2.2. Safe Case 2

An application that executes in module memory and does not use any library is safe from bus fault. (This environment is common for content-only applications such as Handspring's Tiger Woods game module.) When the module is removed, a bus fault is generated when the CPU tries to fetch and execute the next instruction code from the memory module. The bus error trap causes control to be passed to the system bus error handler. The system does not force a soft reset because no libraries were opened by that application. If the Setup application was installed, the system then sends a "remove" message to the Setup application to allow proper clean up. The system then returns control to the "Launcher" application.

Application Environments That Are Vulnerable To Soft Reset

2.3. Vulnerable Case 1:

Applications or Libraries Accessing The Module's Memory Or Hardware

Application or library software that accesses the Springboard memory space should protect its accesses from bus fault. If not protected, the bus fault will trigger the Handspring system event handler determine whether it must force soft reset to avoid corrupted system environment due to an opened library. Thus application or library software must be designed to protect module accesses using Handspring's Try/Catch blocks (HsCardErrTry/HsCardErrCatch). The Try/Catch block implementation enables the library to trap the bus fault from the software accesses to a removed Springboard module and enables for graceful recovery allowing the software to close the library. This clean up corrects the system library environment before the system event handler examines the system environment. Figure 1 below summarizes the issue and the resolution.

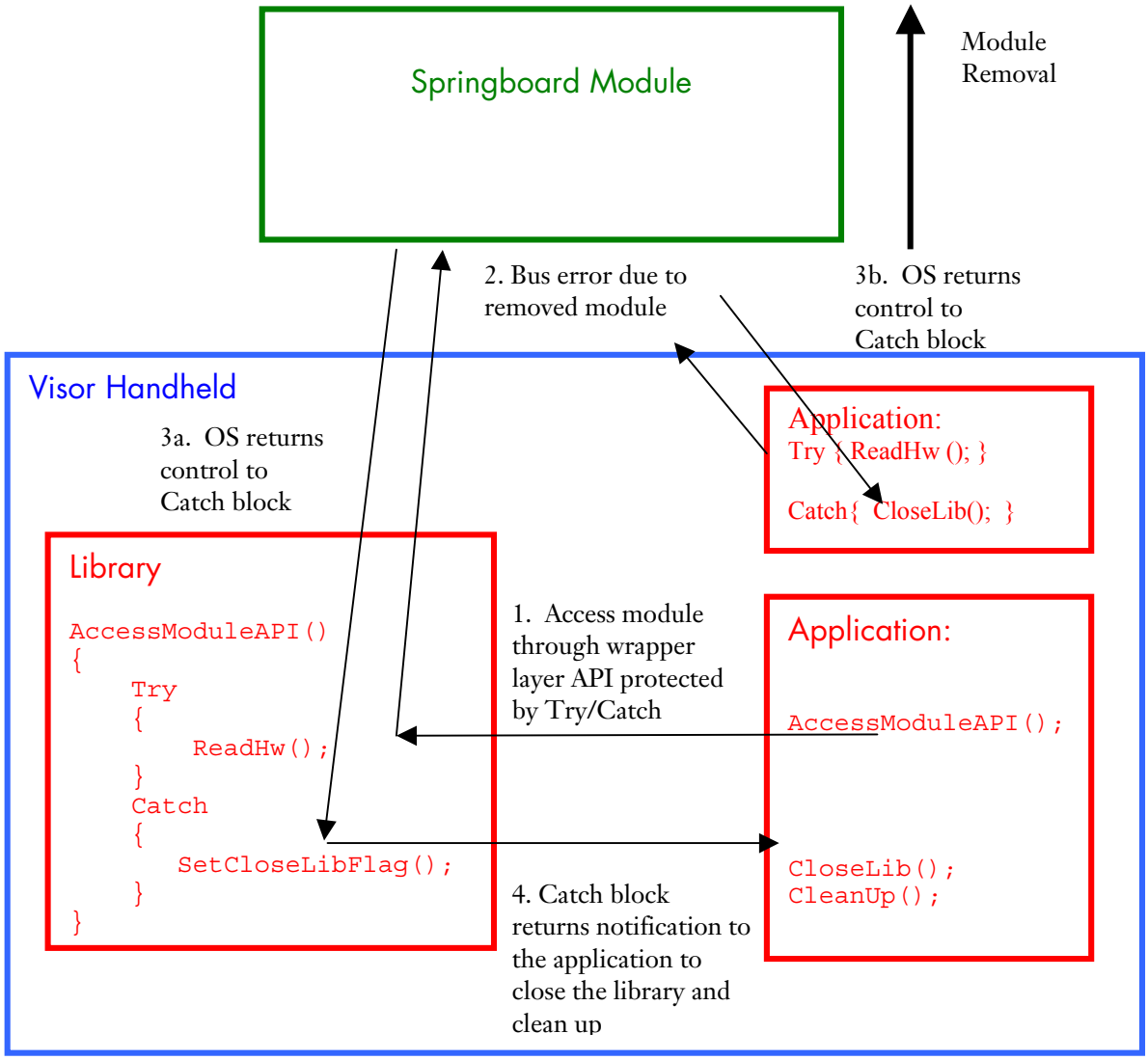

Figure 1. Protecting application or library software that accesses the module

The Try/Catch design can implement one Try/Catch set for each API or one Try/Catch set for a group of similar APIs. Below is some sample code that illustrates both of these approaches.

ModuleLibrary.c

```
AccessModuleAPI()
{
      Err   error = 0;
      volatile    Boolean     needAbort = false;
      volatile    Byte        readbyte;
      volatile    Byte        buffer[ 10 ];

HsCardErrTry
  {
    // Read access of module h/w location
    // ex.  0x28000000 + 0x100000 + 0x5
    readbyte = (* (Byte*) (CardBaseAddr + HwOffsetAddr + Reg5) )
    ReadBuffer( &buffer );
    needAbort = false;
  }

HsCardErrCatch
  {
       // Recover or clean up after a failure in above Try block
       // The code in this block does NOT execute if the above
      //  Try block completes without a card removal

      needAbort = true;
   }
HsCardErrEnd

if (needAbort)
  NotifyAppCloseLib();
return error;

}
```

```
AccessModuleGroupAPI( whichAPI )
{
      Err   error = 0;
      volatile    Boolean     needAbort = false;
      volatile    Byte        readbyte;
      volatile    Byte        buffer[ 10 ];

HsCardErrTry
  {
     // call access of module API
     switch (whichAPI)
       {
          case ReadStatusFlagAPI:
            ReadStatusFlagRegister( StatusFlagRegisterAddr );

          case ClearInterruptStatusFlagAPI:
            ClearInterruptStatusFlagRegister( InterruptStatusFlagRegisterAddr );

          default:
         }
      needAbort = false;
    }

HsCardErrCatch
  {
       // Recover or clean up after a failure in above Try block
       // The code in this block does NOT execute if the above
      //  Try block completes without a card removal

      needAbort = true;
  }
HsCardErrEnd


if (needAbort)
  NotifyAppCloseLib();
return error;

}
```

```
DWord
PilotMain (Word cmd, Ptr cmdPBP, Word launchFlags)
{
  DWord err;

  HsCardErrTry
  {
      // Global protection block for all Springboard accesses.
      // PilotMainBody (cmd, cmdPBP, launchFlags);
  }

  HsCardErrCatch
  {
       // Recover or clean up after a failure in above Try block
       // The code in this block does NOT execute if the above
       // Try block completes without a card removal

      // Close the library
      CloseLib();
    }
```

2.4. Vulnerable Case 2:

Springboard Module Memory Resident Applications Using Libraries

Applications executing directly from the Springboard module memory must implement the Try/Catch protection block in internal RAM so that the system can execute the block even after the Springboard module has be removed. Caching the Try/Catch protection block can be designed using two different implementation schemes.

The first scheme places the application's Try/Catch protection block in the Setup application. (Reminder that the Setup application will be copied from module memory to internal RAM.) The application will call into the Try/Catch protection hook function, and the hook function will call back into the application's PilotMain. Thus the entire application is protected by the Try/Catch block. This scheme requires that the application and the Setup application are intrinsically aware of the other's library management. This scheme would be appropriate for applications designed to centralize all the opened libraries protection into an application controlled by the Springboard module designer. Figure 2 below summarizes the issue and the resolution.

The second scheme places the application's Try/Catch protection block in the application and copies the Try/Catch protection block into dynamically allocated internal RAM. The application will call into the Try/Catch protection hook function, and the hook function will call back into the application's PilotMain. Thus the entire application is protected by the Try/Catch block. This scheme would be appropriate for independent applications designed to manage its own library environment.

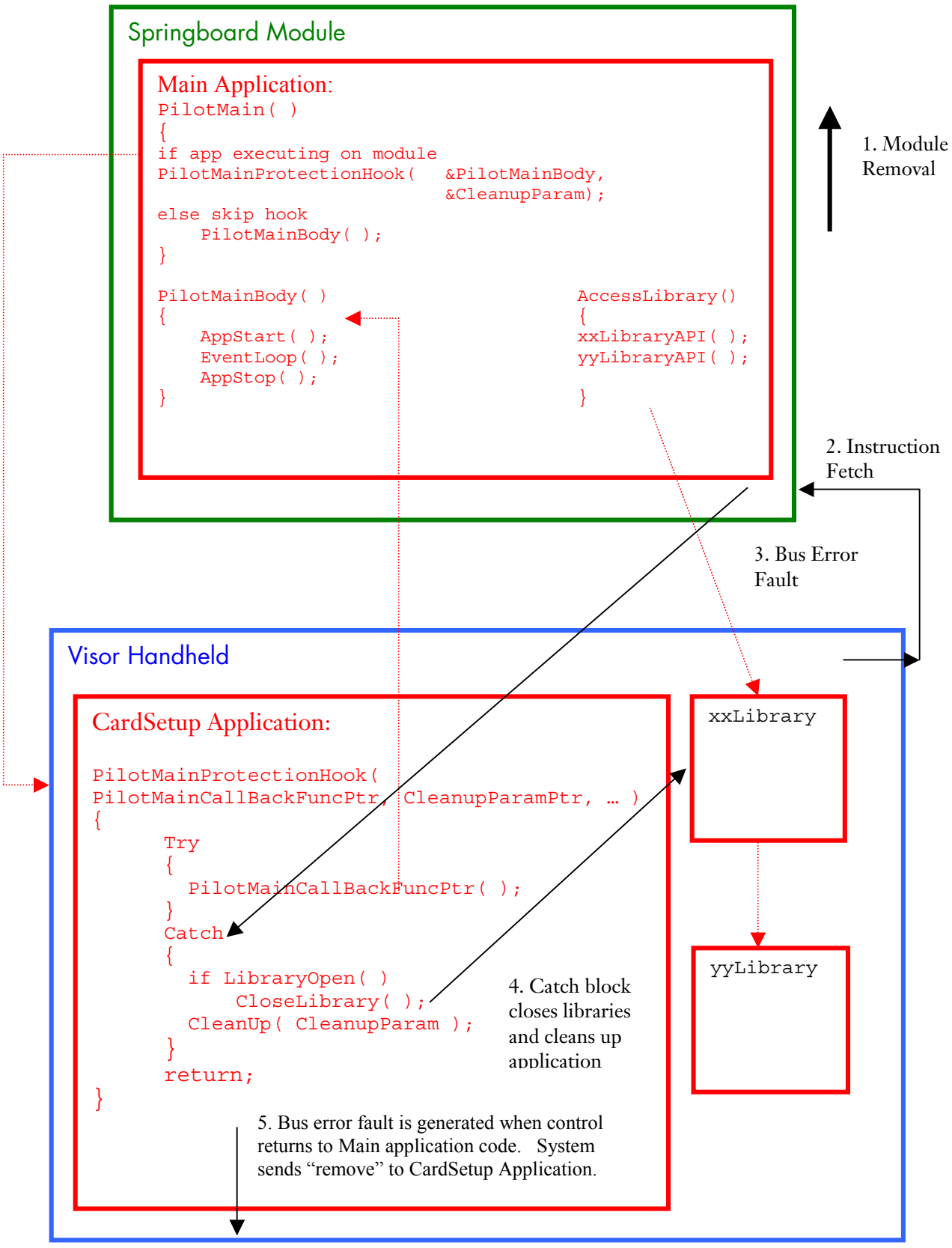

Figure 2. Protecting module-resident applications from open libraries

The sample code below shows the framework for the implementing the Try/Catch protection block in the Setup application. The code is intended only to illustrate the concept; it is not complete.

```
{
  Err   err = 0;
  HsDrcGlobalsType*       gP = 0;

  HsCardErrTry
  {
      // Protect with Try block so that the bus error fault would be caught
      // in the below Catch block

      // Call back application's PilotMain body
      procP();
  }

  HsCardErrCatch
  {
      // Recover or clean up after a failure in above Try block
      // The code in this block does NOT execute if the above
      // Try block completes without a card removal

      gP = (HsDrcGlobalsType*)  CleanUpParamP;

      // Close open libraries
      // Check if the serial library has been un-installed or closed.
      if (gP && gP->serLibRefNum)                     // not un-installed
        if ((SysLibTblEntry (gP->serLibRefNum))->globalsP)        // not closed
            err = SerClose (gP->serLibRefNum);

  }
  HsCardErrEnd

  return err;
}
```

2.5. Vulnerable Case 3:

Applications Using Model Dialogs

Handspring extensions version 1.00 to 1.03 in Visor Palm OS 3.1 devices contain a problem where modal dialog processing blocks the application's Try/Catch protection code from running when the module is being removed. Later Handspring extension versions have corrected this issue. Thus application should check for the appropriate Handspring extension version to implement this protection.

This scenario includes all applications since the "Find" application's modal dialog or the "Datebook" application's alert modal dialog can be triggered at anytime. The problem is caused by the modal dialog's event loop code. When the modal dialog's event loop executes, it displaces the code context of the previous application. This displacement causes the modal dialog's event handler code to execute before the application's Try/Catch protection code

Thus when the module is removed, the system event handler will detect the open library environment and force a soft reset. The application's Try/Catch protection code did not get the opportunity to close the libraries. The resolution is to patch the system event handler to alter the execution sequence so that the either the Try/Catch protection code or some other clean up code is executed. The system event handler patch will to close the libraries upon detection of module removal.

System patches like the system event handler should be implemented in the Setup application (to exist in internal RAM). The application designer must be careful to implement proper logic so that control properly passes back to the previous system event handler and the patching is undone correctly. The un-patching should be done immediately upon detection of module removal.

Once libraries is closed this way, the application must not make any more accesses to those libraries. The application should un-install the library in its clean up code.

Figure 3 below shows a system event handler chain.

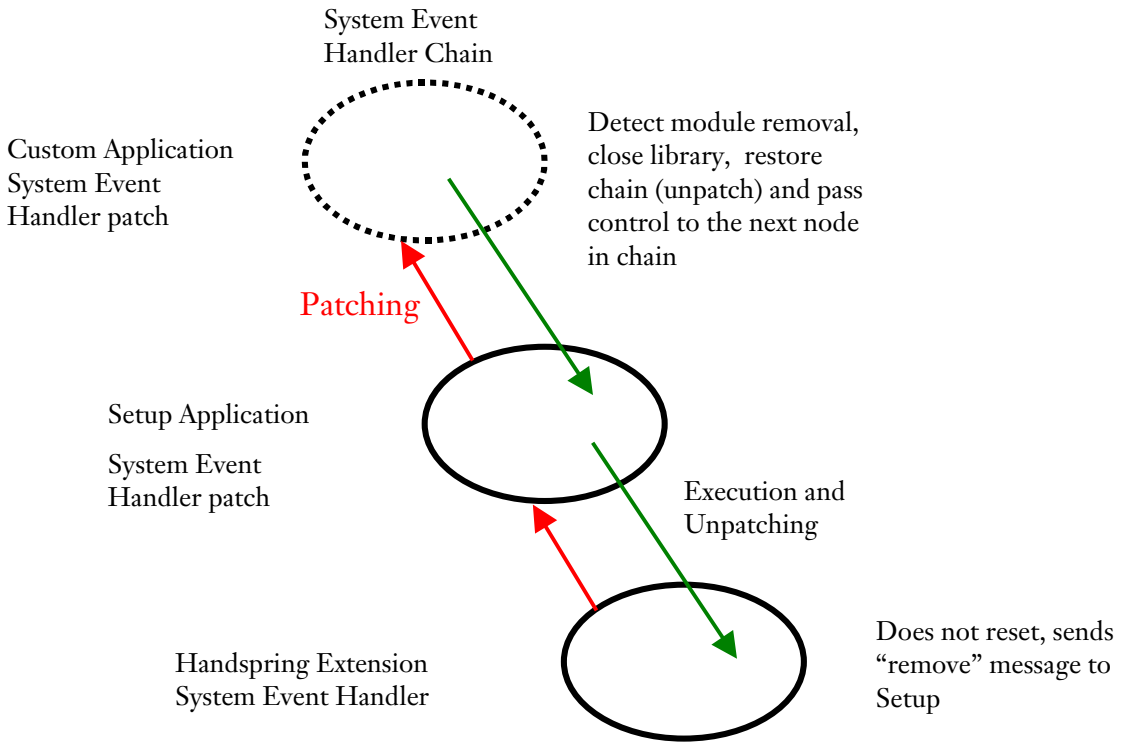

Figure 3. System Event Handler Chain

The sample code below shows the framework for the system handler event patching. The code is intended only to illustrate the concept; it is not complete.

```
Install( )
{
  Boolean   ModulePresent;
      ……

  // For SysHandleEvent, check whether we're running under the version of
Handspring
  // Extensions that needs this patch
  // Ver 1.00 needs patch
  // Ver 1.01 needs patch
  // Ver 1.02 does not exist
  // Ver 1.03 needs patch

  if (sysGetROMVerMajor(hsExtVersion) == 1
            && sysGetROMVerMinor(hsExtVersion) == 0
            && sysGetROMVerFix(hsExtVersion) < 3)
      {
      // Check to determine is module present
      HsCardAttrGet ( 1 /*cardNo*/, hsCardAttrHwInstalled, & ModulePresent);

      // If module is present, patch application's system handler event
      if ( ModulePresent )
              {
              // ------------------------------------------------------------
              // Install our SysHandleEvent patch so that the System Event
              // Handler can be detect card removal and close any libraries
              // ------------------------------------------------------------
              err = HsCardPatchInstall (sysTrapSysHandleEvent,
      (void*)PrvSysHandleEvent);
              ErrNonFatalDisplayIf (err, "patch install err");

              // Set this so PrvRemovePatches will know whether to
              //  remove the patch
              gP->sysHandleEventPatched = true;   // (***need to add this field
      to the
                                                  //  HsDrcGlobalsType structure***)
              }
      }
      ……
}
Remove(Word cardNo)
{     ……

      // Remove the "SysHandleEvent" patch if it was installed
      if (gP->evtGetEventPatched)
              {
              err = HsCardPatchRemove (sysTrapSysHandleEvent);
              ErrNonFatalDisplayIf (err, "patch remove err");

              gP->sysHandleEventPatched = false;
              }
      ……
}
```

```
static Boolean PrvSysHandleEvent(EventPtr eventP)
{
  Err err = 0;
  Boolean handled;
  SysHandleEventFuncType * OldSysHandleEventPtr = 0;
  Boolean ModulePresent = false;

  CALLBACK_PROLOGUE()

  // Todo:
  // This API takes a lot of overhead.  System performance will decrease
  // because SysHandleEvent() is called often.  Need to optimize by caching
  // the old function pointer.  (Global variables are not allowed in Patches
  // because they can be call outside of this application thus rendering the
  // globals out of context.)

  HsCardPatchPrevProc (sysTrapSysHandleEvent, &OldSysHandleEventPtr);

  // If the module is still plugged in, continue as normal; otherwise,
  // close the library and prepare to exit the application.
  HsCardAttrGet(1, hsCardAttrHwInstalled, &ModulePresent);
  if (!ModulePresent)
      {
        HsDrcGlobalsType*       gP = 0;
        Word      cardNo = 1;

        HsCardAttrGet (cardNo, hsCardAttrCardParam, &gP);

        // Close the library
        // Check if the serial library has been un-installed or closed.
        if (gP && gP->serLibRefNum)                          // not un-installed
            if ((SysLibTblEntry (gP->serLibRefNum))->globalsP)    // not closed
              err = SerClose (gP->serLibRefNum);

        // Calling HsCardPatchRemove() will cause a bus error
        // which will jump to the Catch block if one is installed.
        err = HsCardPatchRemove (sysTrapSysHandleEvent);
        ErrNonFatalDisplayIf (err, "patch remove err");

        gP->sysHandleEventPatched = false;
      }

  handled = (OldSysHandleEventPtr)(eventP);

  CALLBACK_EPILOGUE()

  return handled;
}
```

3. History

Date	Revision #	Description of changes
11 Dec 00	2.01	Reformat.
7 Nov 00	2.00	Qualify problem for specific Handspring extension version. Steamline problem description and resolution. Added working sample code.
7 Mar 00	1.00	Initial release.