Microsoft

, Software Design Engineer
Systems Developer Relations

Microsoft Windows Version 3.1

30 March, 1992

Please note that this document and any associated sample code is not officially supported by Microsoft. The information provided here is intended to help software developers programming in the Microsoft Windows environment. Reader are responsible for finding and funding their own support. *Please do not ask the author for technical support as such requests will simply be referred to a support organization such as Microsoft OnLine or Compuserve.*

Updates and error lists to the document and sample code will be posted on both OnLine and Compuserve as necessary.

Your feedback is a very important part in providing documents such as this to the developer community for Microsoft Windows. At the very least, please tell me your impressions. If you can take the time, let me know how you used this document, how you used the sample code, what aspects you found helpful, and what you didn't like. A work like this document is always open to improvement, so please report any problems, errors, or general criticisms you might have. Reach me through mail, fax (dial (206)93MSFAX), or electronic mail at the following addresses:

Internet: kraigb@microsoft.com
Compuserve: 70750,2344

You deserve the best information we can provide. With your help, future documents and samples covering Microsoft Windows technologies will be even better!

Kraig Brockschmidt

Redmond, Washington USA

Bulletproof Functions with ToolHelper

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

KEB.SDR.003

Table of Contents

1 Application Robustness and Fault Trapping

When Windows 3.0 moved applications into protect mode, those applications that attempted to read to or write from memory outside their address space caused a general protection violation. This led to the Unrecoverable Application Error (UAE). Under Windows prior to version 3.0 that only ran in real mode, these applications had no *apparent* problem since they could write to or read from any memory address. However, they were generally trashing memory and possibly causing problems in other applications.

A side-effect of UAEs on Windows 3.0 was system instability–sometimes a UAE in an application could simply lock up the entire system. Windows 3.1 is much improved over its predecessor so that now a UAE will almost never cause a system crash, and some UAEs are not even fatal to an application (in this case, the user may attempt ignore the problem and continue, which does not always insure recovery).

Windows 3.1 has gone through tremendous internal revision to validate every pointer through which it attempts to read or write. In short, Windows 3.1 will not crash inside a Windows API call. However, when an application faults in its own code, there is no salvation–often the user is forced to close the application.

System robustness can only prevent a faulting application from crashing the system; it does nothing to protect the data in that application. This document will describe a method through which a retail application can trap both Divide By Zero exceptions and General Protection (GP) Faults and recover from them using the **Catch** and **Throw** Windows API. Instead of forcing the user to terminate the application on either exception, the application can quietly recover from the problem and fail the operation, if at all possible.

The ability to trap exceptions gives and application considerable power and considerable responsibility. ToolHelper allows you to trap all exceptions, even Int 3 (debug breakpoints) and the Ctrl+Alt+SysRq key. While use of such exception handling (that is, for debugging interrupts) should never be used in a retail product, they can allow your application's debugging version to perform small cleanup, log errors, etc., without user intervention and without terminating the application.

2 The Windows 3.1 ToolHelper DLL

An application can trap exceptions using the **InterruptRegister** and **InterruptUnRegister** functions contained in the Windows 3.1 TOOLHELP.DLL. This library of 34 functions primarily allows tool vendors with debuggers and development environments to handle exceptions and to retrieve a great deal of information about the system without knowing any internal structures. However, you can view your application as having a small built-in function debugger in the form of an exception handler. If an exception happens within a single *function*, your application can take appropriate steps to correct it, much like you might use your debugger to manually clear such a problem.

The Microsoft Windows Software Development Kit version 3.1 contains a sample application (samples\toolhelp\thsample) that demonstrates use of all ToolHelper functions as well as handling all possible interrupts with InterruptRegister. It does not, however, demonstrate the use of Catch and Throw for recovery purposes.

3 InterruptRegister

ToolHelper's InterruptRegister function installs an application-defined callback function that ToolHelper links into the chain of interrupt handlers. This handler is given extraordinary power, as all system interrupts are passed through it. An important implication of this power is that if your handler does not want the interrupt, *make the default processing as fast as possible.*

A bug in the Windows 3.10 version of ToolHelper effectively limits the number of registered handlers to 15, simply because ToolHelper is running out of heap space. Even through the problem will be fixed in a future maintenance upgrade (such as Windows 3.1a), be sure to detect when InterruptRegister fails, and notify the user of the condition (possibly terminating your application). You do not want to execute functions that expect an exception handler to exist. In short, you should treat interrupt handlers as a scarce system resource in the same way you treat the timer.

InterruptRegister takes two parameters: a task handle and a procedure-instance address (from **MakeProcInstance**) to your exported handler function. The task handle identifies the task *registering* the handler and **in no way causes ToolHelper to filter exceptions before passing them on to your handler**. In other words, your handler always receives interrupts for any task, regardless of this task handle passed to InterruptRegister. Applications pass NULL for the task handle, which instructs ToolHelper to use the current task (from **GetCurrentTask**).

For example, you might install an exception handler that is active for the life of the application while processing your main window's WM_CREATE message. If InterruptRegister fails, you can notify the user and terminate the application:

```
case WM_CREATE:
   /*
    * pGlob points to global variables.  We store the value
    * from MakeProcInstance in pfnInt.  Earlier we initialized
    * the hInst field with the instance of the application.
    */

   //ExceptionHandler is the callback's name (HANDLER.ASM)
   pGlob->pfnInt=MakeProcInstance((FARPROC)ExceptionHandler, pGlob->hInst);

   if (NULL!=pGlob->pfnInt)
      {
      //If we fail to register, then fail the create and the application.
      if (!InterruptRegister(NULL, pGlob->pfnInt))
         {
         MessageBox(hWnd, "InterruptRegister Failed.", "Fatal Error", MB_OK);
         return -1L;
         }
      }
   break;
```

4 InterruptUnRegister

InterruptUnRegister is, of course, the opposite function from InterruptRegister, notifying ToolHelper that the application no longer needs the interrupt handler. InterruptRegister simply takes the same task handle as was passed to InterruptRegister. If you registered the handler while processing WM_CREATE as in the example above, then unregister the interrupt during WM_DESTROY. Don't forget to call **FreeProcInstance** for the callback address:

```
case WM_DESTROY:
   //Remove our exception handler and free the proc address.
   if (NULL!=pGlob->pfnInt)
      {
      InterruptUnRegister(NULL);
      FreeProcInstance((FARPROC)pGlob->pfnInt);
      }

   PostQuitMessage(0);
   break;
```

5 Application-Handled Faults

An application can safely handle Divide By Zero exceptions or General Protection (GP) Faults caused by the application's code–*within an application's exception handler you should only deal with those exception that you know were caused in your application*. Under Windows 3.0 and 3.1, where a task switch cannot occur within the scope of a function (unless you explicitly yield control through PeekMessage or GetMessage), you can safely process either of these exceptions--they had to be caused by your code.

Most of the time your application is running, it will not need to handle any exceptions. Within particular functions, it may want to trap Divide by Zero, GP Faults, or both. A single global variable (or a function to set that variable) shared between the application functions and the exception handler is sufficient. Before executing any code that might fault, the application loads this value with bits indicating which exceptions to trap. When an interrupt occurs, the handler checks this value against the actual interrupt that occurred. If there is a match, then the handler can process the fault; otherwise, it passes the interrupt to the next handler in the chain.

6 The Interrupt Handler

Your exception handler is best written in assembly language, since you must move the contents of AX into DS (to access global variables) and because information necessary for processing the interrupt is held on the stack. The most important value is at [SP+06h] which is the interrupt number:

Interrupt	Definition
0	Divide by zero
1	Debugger interrupt
3	Breakpoint
6	Invalid opcode
12	Stack fault

13	GP Fault
14	Page fault not cause by normal virtual memory management
256	Ctrl+Alt+SysRq was pressed.

See the listing for HANDLER.ASM later in this document for a complete description of the values on the stack.

The high bit of the interrupt number might be set, meaning that the fault occurred because of a low-stack condition. This situation requires the application to either terminate itself or pass the interrupt on. Do not attempt to restart the operation or otherwise recover from the error.

In short, applications should only handle Interrupt 0 and 13; all others, including low-stack faults, should be used only in a debugging version of the application.

```
cProc   ExceptionHandler, <PUBLIC,FAR>
cBegin  nogen

        mov     ds,ax           ;Make sure we can reference our data.

        mov     ax,bp           ;Load the interrupt number without changing
        mov     bp,sp           ;the BP register and without using the stack.
        mov     bx,[bp+06]
        mov     bp,ax

        ;
        ; Cycle through the possible faults we are looking for.
        ; First, the high bit might be set in the interrupt meaning
        ; that we have a low-stack fault.  Since we do not handle
        ; these, we exit this handler.
        ;

        test    bx,08000h       ;Check high bit
        jnz     EHExit          ;Leave it it's set--we can't handle it.

        mov     ax,_wException  ;Get the exceptions we want to trap.
        or      ax,ax           ;Do we want any?
        jz      EHExit

EHDivideByZero:
        ;;Check for divide by zero.

        test    ax,EXCEPTION_DIVIDEBYZERO
        jz      EHGPFault
        or      bx,bx
        jz      EHThrow

EHGPFault:
        ;;Check for a GP fault.

        test    ax,EXCEPTION_GPFAULT
        jz      EHExit
        cmp     bx,13
        je      EHThrow

;
; Code for each handling method goes here.  See below.
;

cEnd    nogen
```

Within this function you use one of four methods to handle the exception:

1. Pass the interrupt to the next handler.
2. Terminate the application.
3. Correct the problem and continue.
4. Abort the operation and return an error to the faulting function using Catch and Throw.

Each of these methods is described below; we pay close attention to using Catch and Throw as that method is applied in the sample program given in this document.

7 Pass the Interrupt to the Next Handler

If the exception handler determines that it cannot process the interrupt, it should always pass it to the next handler in the chain by executing a RETF instruction:

```
EHExit:
    retf
```

8 Terminate the Application

If a problem is so critical that there is no hope of continuing the application, then the exception handler can call **TerminateApp** (in TOOLHELP.DLL). This is the default behavior for faulting applications:

```
EHTerminate:
    add     sp,10               ;Clear the return data on the stack.

    ;
    ; With TerminateApp you have the choice to display the standard
    ; UAE box or not.  This sample does since it otherwise would give
    ; no indication of the problem.
    ;
    cCall   TerminateApp, <0, UAE_BOX>
```

Note that you must clear 10 bytes off the stack before calling TerminateApp to insure that the proper stack frame exists at SP. You may also pass NO_UAE_BOX to TerminateApp to suppress the usual box-o-death in case you use another method to notify the user.

9 Correct the Problem and Continue

If your application maintains state information about what it was doing at the time of a fault, then it might be possible to correct the problem (by changing a pointer or a terminating condition) and continue the operation. This is similar to the Windows 3.1 UAE box that might allow you to ignore a GP Fault. In this case you correct the problem, clean up 10 bytes on the stack (to insure the proper stack frame at SP) and execute and IRET instruction:

```
EHRestart:
    ;
    ; This code illustrates what we need to do if we wanted to
    ; clean up the problem and restart the instruction that faulted.
    ;
```

```
add     sp,10               ;Clear the return data on the stack.
iret                        ;Return to the faulting instruction.
```

10 Abort the Operation with Catch and Throw

The most powerful and the most dangerous method to handle an exception is to use **Catch** and **Throw** to essentially execute a far **goto**; Catch and Throw are the Windows equivalents of the C run-time **setjmp** and **longjmp** functions.

When entering a function that might fault, call Catch to save the register state in a global CATCHBUF structure. The first time you call Catch here it will always return zero, signifying that no error occurred. Within your exception handler you call Throw, passing to it the same CATCHBUF used in Catch and a non-zero value. This value is *again* returned from the original Catch call. Simply said, Throw reinstates the registers saved in the first call to Catch, which causes Catch to return *again*, within the same function as it already returned once. However, this time it returns with a non-zero value so you know there was an error.

For example, the code below calls Catch and watches for a GP Fault. When Catch returns with a non-zero value, we attempt to perform any necessary cleanup and return a failure code:

```
WORD       i;
HANDLE     hMem;

/*
 * Call Catch and indicate what exceptions to trap.
 *
 * The first time we call Catch here we will get a 0 return value.
 * When we call Throw in our exception handler, Catch returns with
 * the value given in the second parameter to Throw.  Throw must
 * use the same CATCHBUF we fill here in order to return here.
 */

//Indicate the trap(s) we want.
wException=EXCEPTION_GPFAULT;

/*
 * Save the register state.  pcbEx is a global pointer to a CATCHBUF structure,
 * then check if we returned from the exception handler or Catch itself.
 */
if (0!=Catch(pcbEx))
   {
   /*
    * Free any resources this function allocated, perform other
    * cleanup, turn OFF any exception handling, and return a failure.
    */

   wException=EXCEPTION_NONE;
   return NULL;
   }

//The actual function goes here.
...
```

The call to Throw within the exception handler is trivial:

```
EHThrow:
      ccall   Throw,<_pcbEx, 1>  ;Return the exception to the faulting
                                 ;function.
```

µ §

Be extremely careful when using Catch and Throw in an application. Only make jumps using Throw within the scope of a function. You can create very unstable situations by calling Catch once and calling Throw much later after your application has processed messages or there has been a task switch. With Catch and Throw, you must be sure that the fault was caused by your application's code, otherwise you may not insure that other code is cleaning up properly.

Also note that recovering from the error after Catch returns non-zero may not be possible. Consider a situation where a fault occurs and you would like to save the user's data. If the operation that crashed *was* in saving a file, then you will probably fault again which could put your application in a much worse state.

Catch and Throw give you a lot of power–be careful not to abuse it.

11 Sample Application: FAULT.EXE

MAKEFILE

```
#
# MAKEFILE
#
# FAULT
# Copyright(c) Microsoft Corp. 1992 All Rights Reserved
#


#Remove '#' from next line for "silent" operation
!CMDSWITCHES +s
```

```
#Compiler and assembler flags

CFLAGS=-c -G2sw -W3 -AS -Od -Zpe

AFLAGS=-Mx -L


#Standard definitions.

DEFS=

INCLS=fault.h

OBJS = fault.obj handler.obj

RCFILES=fault.h fault.ico


.SUFFIXES: .h .c .asm .obj .exe .cxx .res .rc .bas


goal:  fault.exe


##### Rules


.asm.obj:

    masm $(AFLAGS) $(DEFS) $*.asm;


.c.obj:

    cl $(CFLAGS) $(DEFS) $*.c



.rc.res:
```

```
    rc -r $*.rc



fault.exe : $(OBJS) fault.res fault.lnk fault.def
    link @fault.lnk
    rc -v fault.res



##### Dependencies #####
handler.obj  : handler.asm
fault.obj    : fault.c    $(INCLS)
fault.res    : fault.rc   $(INCLS) $(RCFILES)
```

FAULT.H

```
/*
 * FAULT.H
 *
 * Definitions and function prototypes for Fault.
 *
 * Copyright(c) Microsoft Corp. 1992 All Rights Reserved
 */
```

```
#define IDR_MENU            1

#define IDM_EXDIVIDEBYZERO  100
#define IDM_EXGPFAULT       101

//Structure holding the "global" variables.  Creating a structure with
typedef struct
   {
   HWND            hWnd;              //Top-level application window.
   HANDLE          hInst;             //Application instance handle.
   FARPROC         pfnInt;            //GP Fault Handler thunk.
   } GLOBALS;

typedef GLOBALS FAR * LPGLOBALS;

//External:
extern LPGLOBALS     pGlob;
extern CATCHBUF      cbEx;
extern LPCATCHBUF    pcbEx;
extern WORD          wException;

/*
 * Flag values to store in the wException global variable that tells
```

```
 * the handler what type of exceptions we specifically want.
 */

#define EXCEPTION_NONE          0x0000      //Turns handling off.
#define EXCEPTION_DIVIDEBYZERO  0x0001
#define EXCEPTION_GPFAULT       0x0002
#define EXCEPTION_ALL           0x0003      //Looks for all exceptions above.

/*
 * Function prototypes.
 */

//FAULT.C
LONG     FAR PASCAL FaultWndProc(HWND, UINT, UINT, LONG);
BOOL     FAR PASCAL FPerformCalculation(void);
HANDLE   FAR PASCAL HAllocateNumbers(void);

//HANDLER.ASM
void     FAR PASCAL ExceptionHandler(void);
```

FAULT.C

```
/*
 * FAULT.C
 *
 * Very small Windows application demonstrating how to use the TOOLHELP
 * library to trap GP Faults and Divide by Zero exceptions.  Trapping these
 * faults allows an application to perform cleanup, save files, and
 * otherwise insure integrity of the user's data.
 *
  * Copyright(c) Microsoft Corp. 1992 All Rights Reserved
 *
 */

#include <windows.h>
#include <toolhelp.h>
#include "fault.h"



//Global variable block.
GLOBALS     stGlobals;
LPGLOBALS   pGlob=&stGlobals;


/*
 * These global variables hold information that is needed from the
 * interrupt handler.  They are set apart here to make them more visible.
```

```
 */
CATCHBUF    cbEx;                   //Stores register state.
LPCATCHBUF  pcbEx=(LPCATCHBUF)&cbEx;    //Convenient pointer
WORD        wException;             //Indicates which exceptions to trap.



/*
 * WinMain
 *
 * Purpose:
 *  Main entry point of application.   Should register the app class
 *  if a previous instance has not done so and do any other one-time
 *  initializations.
 *
 * Parameters:
 *  Standard
 *
 * Return Value:
 *  Value to return to Windows--termination code.
 *
 */


int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
```

```
{
WNDCLASS        wc;
MSG             msg;

pGlob->hInst=hInstance;



if (!hPrevInstance)
    {
    /*
     * Note that we do not need to unregister classes on a failure
     * since that's part of automatic app cleanup.
     */
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = FaultWndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = pGlob->hInst;
    wc.hIcon         = LoadIcon(NULL, IDI_HAND);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = COLOR_APPWORKSPACE + 1;
    wc.lpszMenuName  = MAKEINTRESOURCE(IDR_ICON);
    wc.lpszClassName = "Fault";
```

```
    if (!RegisterClass(&wc))
        return FALSE;
    }

pGlob->hWnd=CreateWindow("Fault", "Exception Handler",
                    WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT, 400, 120,
                    NULL, NULL, hInstance, NULL);

if (NULL==pGlob->hWnd)
    return FALSE;

ShowWindow(pGlob->hWnd, nCmdShow);
UpdateWindow(pGlob->hWnd);

while (GetMessage(&msg, NULL, 0,0 ))
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }

return msg.wParam;
}
```

```
/*
 * FaultWndProc
 *
 * Purpose:
 *  Window class procedure.  Standard callback.
 *
 * Parameters:
 *  The standard.
 *
 * Return Value:
 *  The standard.
 *
 */
```

```
long FAR PASCAL FaultWndProc(HWND hWnd, UINT iMsg, UINT wParam, LONG lParam)
   {
   HANDLE      hMem;


   switch (iMsg)
      {
      case WM_CREATE:
         /*
          * Install the ExceptionHandler function in HANDLER.ASM
          * as our fault handler and store the proc address in the
          * global variable block.
          */


         pGlob->pfnInt=MakeProcInstance((FARPROC)ExceptionHandler, pGlob->hInst);


         if (NULL!=pGlob->pfnInt)
            {
            //If we fail to register, fail the create and the application.
            if (!InterruptRegister(NULL, pGlob->pfnInt))
               {
               MessageBox(hWnd, "InterruptRegister Failed.", "Fatal Error", MB_OK);
               return -1L;
               }
```

```
            }
        break;



    case WM_DESTROY:
        //Remove our exception handler and free the proc address.
        if (NULL!=pGlob->pfnInt)
            {
            InterruptUnRegister(NULL);
            FreeProcInstance((FARPROC)pGlob->pfnInt);
            }

        PostQuitMessage(0);
        break;




    case WM_COMMAND:
        switch (wParam)
            {
            case IDM_EXDIVIDEBYZERO:
                if (FPerformCalculation())
                    {
```

```
            //This should NOT happen.
            MessageBox(hWnd, "IMPOSSIBLE: Missed a Divide by Zero!",
                    "Fault", MB_OK | MB_ICONHAND);
            }
        else
            {
            MessageBox(hWnd, "Calculation failed: Divide by zero.",
                    "Fault", MB_OK | MB_ICONEXCLAMATION);
            break;
            }

        break;


    case IDM_EXGPFAULT:
        hMem=HAllocateNumbers();

        if (NULL!=hMem)
            {
            //This should NOT happen.
            MessageBox(hWnd, "IMPOSSIBLE:  Missed the GP Fault!",
                    "Fault", MB_OK | MB_ICONHAND);

            GlobalFree(hMem);
```

```
                }
            else
                {
                MessageBox(hWnd, "Allocation failed: GP fault.",
                        "Fault", MB_OK | MB_ICONEXCLAMATION);
                break;
                }

            break;


          default:
            break;
          }
        break;


    default:
        return (DefWindowProc(hWnd, iMsg, wParam, lParam));
    }

return 0L;
}
```

```
/*
 * FPerformCalculation
 *
 * Purpose:
 *  Attempts to divide the number 10000 by the numbers 1 through 6.
 *  However, this function was poorly written to use the wrong
 *  terminating condition in a while loop, so the loop executes with
 *  zero as the divisor and faults.
 *
 * Parameters:
 *  None
 *
 * Return Value:
 *  BOOL        TRUE if the function succeeded (this should not happen)
 *              FALSE if the function failed, even on a divide by zero
 *              exception.
 */

BOOL FAR PASCAL FPerformCalculation(void)
   {
   WORD       i;
   WORD       wValue;
```

```
/*
 * Call Catch and indicate what exceptions to trap.
 *
 * The first time we call Catch here we will get a 0 return value.
 * When we call Throw in our exception handler, Catch returns with
 * the value given in the second parameter to Throw.  Throw must
 * use the same CATCHBUF we fill here in order to return here.
 */

//Indicate the trap(s) we want.
wException=EXCEPTION_DIVIDEBYZERO;

//Save the register state and check if we returned from the exception handler.
if (0!=Catch(pcbEx))
    {
    /*
     * Now we can safely exit this procedure, skipping the code
     * that faulted.  We indicate that we now want no exceptions
     * by setting wException to EXCEPTION_NONE.
     *
     ***BE SURE to turn off exception handling that uses Catch
     *  and Throw between messages.  In other words, only use
     *  Catch and Throw within the scope of a function, NOT on
```

```
     *  the scope of an application.
     */

    wException=EXCEPTION_NONE;
    return FALSE;
    }


i=6;
wValue=10000;

/*
 * When we check i==1, the condition is TRUE so
 * we continue the loop.  However, the post-decrement
 * on i makes it zero, which will fault.
 */

while (i--)
    wValue /=i;

//We should never get here.
wException=EXCEPTION_NONE;
return TRUE;
}
```

```
/*
 * HAllocateNumbers
 *
 * Purpose:
 *  Attempts to allocate a 1K block of memory and fill it with the
 *  repeating sequence of the values 0 through 255.  However, due to
 *  another bug in this function, we end up writing past the end of
 *  the segment.  We trap the GP Fault and recover by freeing the memory
 *  and indicating that the function failed.
 *
 * Parameters:
 *  None
 *
 * Return Value:
 *  HANDLE          A global memory handle containing the numbers if
 *                  successful, NULL otherwise (including when we GP fault).
 */

HANDLE FAR PASCAL HAllocateNumbers(void)
   {
```

```
LPSTR       psz;
WORD        i;
HANDLE      hMem;

/*
 * Call Catch and indicate what exceptions to trap.
 *
 * The first time we call Catch here we will get a 0 return value.
 * When we call Throw in our exception handler, Catch returns with
 * the value given in the second parameter to Throw.  Throw must
 * use the same CATCHBUF we fill here in order to return here.
 */

//Indicate the trap(s) we want.
wException=EXCEPTION_GPFAULT;

//Save the register state and check if we returned from the exception handler.
if (0!=Catch(pcbEx))
    {
    /*
     * Free any resources this function allocated, perform other
     * cleanup, turn OFF any exception handling, and return a failure.
     */
```

```
    if (NULL!=hMem)
        {
        GlobalUnlock(hMem);
        GlobalFree(hMem);
        }

    wException=EXCEPTION_NONE;
    return NULL;
    }


//Get 1024 bytes of memory.
hMem=GlobalAlloc(GMEM_MOVEABLE, 1024);
psz=GlobalLock(hMem);

/*
 * Write to 1025 bytes of memory, thus accidentally walking over
 * the edge.  Another case where an erroneous terminating condition
 * in a loop can cause such a problem.
 */

i=0;
while (i++ <= 1024)     //Should be i++ < 1024, not <=
    *psz++=(char)i;
```

```
    //We should never get here to return the handle.
    GlobalUnlock(hMem);
    wException=EXCEPTION_NONE;
    return hMem;
    }
```

FAULT.INC

```
;
; FAULT.INC
;
; Definitions and external references for use from assembly modules.
;
; Copyright(c) Microsoft Corp. 1992 All Rights Reserved
;




;; External data.
externFP <Throw>                ;From Kernel
externFP <TerminateApp>         ;From ToolHelp
```

```
externW  <_wException>          ;Indicates what exceptions we want.
externFP <_pcbEx>               ;Pointer to a CATCHBUF to use in Throw


;; Bits for wException.  These must match FAULT.H.
EXCEPTION_NONE          equ     00h     //Turns handling off.
EXCEPTION_DIVIDEBYZERO  equ     01h
EXCEPTION_GPFAULT       equ     02h
EXCEPTION_ALL           equ     03h     //Looks for all exceptions above.
```

HANDLER.ASM

```
;
; HANDLER.ASM
;
; ExceptionHandler function used as the callback for the GP Fault Handler.
;
; Copyright(c) Microsoft Corp. 1992 All Rights Reserved
;


    .xlist
```

```
    ?PLM=1
    ?WIN=1
    include cmacros.inc
    include toolhelp.inc
    include fault.inc
    .list



createSeg HANDLER_TEXT, HANDLER_TEXT, BYTE, PUBLIC, CODE
sBegin    HANDLER_TEXT
assumes   CS,HANDLER_TEXT
assumes   DS,_DATA



;
; ExceptionHandler
;
; Purpose:
;  Exception handling function called from ToolHelp when it detects
;  an exception.  If the error was caused by our application AND we
;  want to trap it, then we use Throw to return control to the function
;  that faulted at a point BEFORE the erroneous code.  If we do not
;  want the fault, then we just pass it to the next interrupt handler.
;
```

```
;  If the fault was caused by our application, we use Throw to return to
;  whatever function caused it.  Otherwise we just let the fault pass.
;
; Parameters:
;  None; however, the stack contains values of interest.  We use AX to
;  simply set the DS register properly.
;
;    |          .          |
;    |          .          |
;    |          .          |
;    |----------------------|
;    |  SS (fault)          |  SP + 12h
;    |----------------------|
;    |  SP (fault)          |  SP + 10h
;    |----------------------|
;    |  Flags (fault)       |  SP + 0Eh
;    |----------------------|
;    |  CS (fault)          |  SP + 0Ch
;    |----------------------|
;    |  IP (fault)          |  SP + 0Ah
;    |----------------------|
;    |  handle (internal)   |  SP + 08h
;    |----------------------|
;    |  interrupt number    |  SP + 06h
```

```
;    |----------------------|
;    |   AX (NOT the DS)    |  SP + 04h
;    |----------------------|
;    |   CS (toolhelp.dll)  |  SP + 02h
;    |----------------------|
;    |   IP (toolhelp.dll)  |  SP + 00h
;    +----------------------+
;
;  Note that the interrupt number may have the high bit set to
;  indicate a low-stack fault (above and beyond a normal Int 12 stack)
;  fault.  We trap this condition if we're trapping a stack fault or a
;  GP fault.
;



cProc   ExceptionHandler, <PUBLIC,FAR>

cBegin  nogen

        mov     ds,ax               ;Make sure we can reference our data.

        mov     ax,bp
        mov     bp,sp
        mov     bx,[bp+06]
```

```
        mov     bp,ax


        ;
        ; Cycle through the possible faults we are looking for.
        ; If BX contains that fault, then we use Throw to return
        ; the error to the application.  Otherwise we pass the
        ; fault down the chain.
        ;
        ; First, the high bit might be set in the interrupt meaning
        ; that we have a low-stack fault.  Since we do not handle
        ; these, we exit this handler.
        ;

        test    bx,08000h           ;Check high bit
        jnz     EHExit              ;Leave it it's set--we can't handle it.

        mov     ax,_wException      ;Get the exceptions we want to trap.
        or      ax,ax               ;Do we want any?
        jz      EHExit

EHDivideByZero:
        ;
        ; Check for divide by zero.
        ;
```

```
        test    ax,EXCEPTION_DIVIDEBYZERO
        jz      EHGPFault
        or      bx,bx
        jz      EHThrow

EHGPFault:
        ;
        ; Check for a GP fault.
        ;

        test    ax,EXCEPTION_GPFAULT
        jz      EHExit
        cmp     bx,13
        je      EHThrow

EHThrow:
        ccall   Throw,<_pcbEx, 1>  ;Return the exception to the faulting
                                   ;function.

EHExit:
        retf

EHRestart:
```

```
        ;
        ; This code illustrates what we need to do if we wanted to
        ; clean up the problem and restart the instruction that faulted.
        ; This exception handler does not use this code.
        ;

        add     sp,10                   ;Clear the return data on the stack.
        iret                            ;Return to the faulting instruction.

EHTerminate:
        ;
        ; This code illustrates what we need to do if we wanted to
        ; terminate the applciation on this exception.  This exception
        ; handler does not use this code.
        ;

        add     sp,10                   ;Clear the return data on the stack.

        ;
        ; With TerminateApp you have the choice to display the standard
        ; UAE box or not.  This sample does since it otherwise would give
        ; no indication of the problem.
        ;
        cCall   TerminateApp, <0, UAE_BOX>
```

```
cEnd    nogen

sEnd    HANDLER_TEXT

        END
```

FAULT.RC

```
/*
 * FAULT.RC
 *
 * Resources such as icons, menus, strings, accelerators, and dialogs.
 *
 * Copyright(c) Microsoft Corp. 1992 All Rights Reserved
 */

#include "fault.h"

IDR_MENU MENU
    BEGIN
     POPUP "&Exceptions"
```

```
    BEGIN
     MENUITEM "&Divide by Zero",   IDM_EXDIVIDEBYZERO
     MENUITEM "&GP Fault",         IDM_EXGPFAULT
    END
  END
```

FAULT.DEF

```
NAME           FAULT
DESCRIPTION    'Exception Handler with Catch and Throw'
EXETYPE        WINDOWS
STUB           'WINSTUB.EXE'

CODE           PRELOAD MOVEABLE DISCARDABLE
DATA           PRELOAD MOVEABLE MULTIPLE

SEGMENTS
               _TEXT       PRELOAD MOVEABLE DISCARDABLE
               HANDLER_TEXT PRELOAD FIXED

HEAPSIZE       8192
STACKSIZE      8192
```

```
EXPORTS        FaultWndProc          @1
               ExceptionHandler      @2
```

FAULT.LNK

```
fault handler
fault /al:16
/map /li
libw toolhelp slibcew/NOD/NOE
fault.def
```