January 1997

MICROSOFT SYSTEMS JOURNAL

A Crash Course on the Depths of Win32™ Structured Exception Handling

At its heart, Win32 structured exception handling is an operating system-provided service. All the docs you're likely to find about SEH describe one particular com-piler's runtime library wrapping around the operating system implementation. I'll strip SEH to its most fundamental concepts.

Matt Pietrek

This article assumes you're familiar with C++, Win32

Code for this article: Exception.zip (6KB)

Matt Pietrek is the author of Windows 95 System Programming Secrets *(IDG Books, 1995). He works at NuMega Technologies Inc., and can be reached at mpietrek@tiac.com.*

Of all the facilities provided by Win32® operating systems, perhaps the most widely used but underdocumented is structured exception handling (SEH). When you think of Win32 structured exception handling, you probably think of terms like _try, _finally, and _except. You can find good descriptions of SEH in just about any competent Win32 book (even the remedial ones). Even the Win32 SDK has a fairly complete overview of using structured exception handling with _try, _finally, and _except.

With all this documentation, where do I get off saying that SEH is underdocumented? At its heart, Win32 structured exception handling is an operating system-provided service. All the documentation you're likely to find about SEH describes one particular compiler's runtime library wrapping around the operating system implementation. There's nothing magical about the keywords _try, _finally, or _except. Microsoft's OS and compiler groups defined these keywords and what they do. Other C++ compiler vendors have simply gone along with their semantics. While the compiler SEH layer tames the nastiness of raw operating system SEH, it's had the effect of keeping the raw operating system SEH details from public view.

I've received numerous email messages from people who have needed to implement compiler-level SEH and couldn't find much in the way of documentation for the operating system facilities. In a rational world, I'd be able to point to the runtime library sources for Visual C++ or Borland C++ and be done with it. Alas, for some unknown reason, compiler-level SEH seems to be a big secret. Neither Microsoft nor Borland provide the source code for the innermost layer of their SEH support.

In this article, I'll strip structured exception handling down to its most fundamental concepts. In doing so, I'll separate what the OS provides from what compilers provide via code generation and runtime library support. When I dive into the code for key operating system routines, I'll use the Intel version of Windows NT® 4.0 as my base. Most of what I'll describe is equally applicable on other processors, however.

I'm going to avoid the issue of true C++ exception handling, which uses catch() instead of _except. Under the hood, true C++ exception handling is implemented very similarly to what I'll describe here. However, true C++ exception handling has some additional complexities that would just cloud the concepts I want to cover.

In digging through obscure .H and .INC files to piece together what constitutes Win32 SEH, one of the best sources of information turned out to be the IBM OS/2 header files (particularly BSEXCPT.H). This shouldn't be too surprising if you've been in this business for a while. The SEH mechanisms described here were defined back when Microsoft was still working on OS/2. For this reason, you'll find SEH under Win32 and OS/2 to be remarkably similar.

SEH in the Buff

Since the details of SEH can be overwhelming if taken all at once, I'll start out small and work my way up through the layers. If you've never worked with structured exception handling before, you're in good shape; you have no preconceived notions. If you've used SEH before, try to clear your head of words like _try, GetExceptionCode, and EXCEPTION_EXECUTE_HANDLER. Pretend that this is all new to you. Take a deep breath. Are you ready? Good.

Imagine I told you that when a thread faults, the operating system gives you an opportunity to be informed of the fault. More specifically, when a thread faults, the operating system calls a user-defined callback function. This callback function can do pretty much whatever it wants. For instance, it might fix whatever caused the fault, or it might play a Beavis and Butt-head .WAV file. Regardless of what the callback function does, its last act is to return a value that tells the system what to do next. (This isn't strictly true, but it's close enough for now.)

Given that the system calls you back when your code makes a mess, what should the callback function look like? In other words, what sort of information would you want to know about the exception? It really doesn't matter because Win32 has made up your mind for you. An exception callback function looks like this:

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

This prototype, which comes from the standard Win32 header file EXCPT.H, looks a little overwhelming at first. If you take it slowly, it's really not so bad. For starters, ignore the return type (EXCEPTION_DISPOSITION). What you basically have is a function called _except_handler that takes four parameters.

The first parameter to an _except_handler callback is a pointer to an EXCEPTION_RECORD. This structure is defined in WINNT.H, shown below:

```
typedef struct _EXCEPTION_RECORD {
DWORD ExceptionCode;
DWORD ExceptionFlags;
struct _EXCEPTION_RECORD *ExceptionRecord;
PVOID ExceptionAddress;
DWORD NumberParameters;
DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
}  EXCEPTION_RECORD;
```

The ExceptionCode parameter is the number that the operating system assigned to the exception. You can see a list of various exception codes in WINNT.H by searching for #defines that start with "STATUS_". For example, the code for the all-too-familiar STATUS_ACCESS_VIOLATION is 0xC0000005. A more complete set of exception codes can be found in NTSTATUS.H from the Windows NT DDK. The fourth element in the EXCEPTION_RECORD structure is the address where the exception occurred. The remaining EXCEPTION_RECORD fields can be ignored for the moment.

The second parameter to the _except_handler function is a pointer to an establisher frame structure. This is a vital parameter in SEH, but for now you can ignore it.

The third parameter to the _except_handler callback is a pointer to a CONTEXT structure. The CONTEXT structure is defined in WINNT.H and represents the register values of a particular thread. Figure 1 shows the fields of a CONTEXT structure. When used for SEH, the CONTEXT structure represents the register values at the time of the exception. Incidentally, this CONTEXT structure is the same structure used with the GetThreadContext and SetThreadContext APIs.

The fourth and final parameter to the _except_handler callback is called the DispatcherContext. It also can be ignored for the moment.

To briefly recap thus far, you have a callback function that's called when an exception occurs. The callback takes four parameters, three of which are pointers to structures. Within these structures, some fields are important, others not so important. The key point is that the _except_handler callback function receives a wealth of information, such as what type of exception occurred and where it occurred. Using this information, the exception callback needs to decide what to do.

While it's tempting for me to throw together a quickie sample program that shows the _except_handler callback in action, there's still something missing. In particular, how does the operating system know where to call when a fault occurs? The answer is yet another structure called an EXCEPTION_REGISTRATION. You'll see this structure throughout this article, so don't skim past this part. The only place I could find a formal definition of an EXCEPTION_REGISTRATION was in the EXSUP.INC file from the Visual C++ runtime library sources:

```
_EXCEPTION_REGISTRATION struc
     prev    dd      ?
     handler dd      ?
 _EXCEPTION_REGISTRATION ends
```

You'll also see this structure referred to as an _EXCEPTION_REGISTRATION_RECORD in the definition of the NT_TIB structure from WINNT.H. Alas, nowhere is an _EXCEPTION_REGISTRATION_RECORD defined, so all I have to work from is the assembly language struc definition in EXSUP.INC. This is just one example of what I meant earlier when I said that SEH was underdocumented.

In any event, let's return to the question at hand. How does the OS know where to call when an exception occurs? The EXCEPTION_REGISTRATION structure consists of two fields, the first of which you can ignore for now. The second field, handler, contains a pointer to an _except_ handler callback function. This gets you a little closer, but now the question becomes, where does the OS look to find the EXCEPTION_REGISTRATION structure?

To answer this question, it's helpful to remember that structured exception handling works on a per-thread basis. That is, each thread has its own exception handler callback function. In my May 1996 column, I described a key Win32 data structure, the thread information block (aka the TEB or TIB). Certain fields of this data structure are the same between Windows NT, Windows® 95, Win32s, and OS/2. The first DWORD in a TIB is a pointer to the thread's EXCEPTION_REGISTRATION structure. On the Intel Win32 platform, the FS register always points to the current TIB. Thus, at FS:[0] you can find a pointer to an EXCEPTION_REGISTRATION structure.

Now I'm getting somewhere! When an exception occurs, the system looks at the TIB of the faulting thread and retrieves a pointer to an EXCEPTION_REGISTRATION structure. In this structure is a pointer to an _except_handler callback function. The operating system now knows enough to call the _except_handler function, as shown in Figure 2.

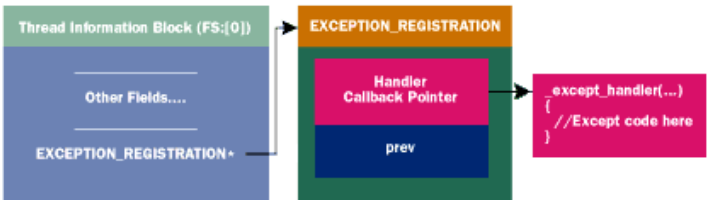

Figure 2 _except_handler function

With the minimal pieces finally put together, I wrote a small program to demonstrate this very simple description of OS-level structured exception handling. Figure 3 shows MYSEH.CPP, which has only two functions. Function main uses three inline ASM blocks. The first block builds an EXCEPTION_REGISTRATION structure on the stack via two PUSH instructions ("PUSH handler" and "PUSH FS:[0]"). The PUSH FS:[0] saves the previous value of FS:[0] as part of the structure, but that's not important at the moment. The significant thing is that there's an 8-byte EXCEPTION_REGISTRATION structure on the stack. The very next instruction (MOV FS:[0],ESP) makes the first DWORD in the thread information block point at the new EXCEPTION_REGISTRATION structure.

If you're wondering why I built the EXCEPTION_REGISTRATION structure on the stack rather than using a global variable, there's a good reason. When you use a compiler's _try/_except syntax, the compiler also builds the EXCEPTION_REGISTRATION struct on the stack. I'm simply showing you a simplified version of what a compiler would do if you used _try/_except.

Back to function main, the next __asm block intentionally causes a fault by zeroing out the EAX register (MOV EAX,0), and then uses the register's value as a memory address, which the next instruction writes to (MOV [EAX],1). The final __asm block removes this simple exception handler: first it restores the previous contents of FS:[0], and then it pops the EXCEPTION_REGISTRATION record off the stack (ADD ESP,8).

Now, pretend that you're running MYSEH.EXE and you'll see what happens. When the MOV [EAX],1 instruction executes, it causes an access violation. The system looks at the FS:[0] in the TIB and finds a pointer to the EXCEPTION_REGISTRATION structure. In the structure is a pointer to the _except_handler function in MYSEH.CPP. The system then pushes the four required parameters (which I described earlier) onto the stack and calls the _except_handler function.

Once inside _except_handler, the code first indicates "Yo! I made it here!" via a printf statement. Next, _except_handler fixes the problem that caused the fault. That is, the EAX register points to a memory address that can't be written to (address 0). The fix is to change the EAX value in the CONTEXT structure so that it points to a location where writing is allowed. In this simple program, a DWORD variable (scratch) has been designated for just this purpose. The last act of the _except_handler function is to return the value ExceptionContinueExecution, which is defined in the standard EXCPT.H file.

When the operating system sees that ExceptionContinueExecution was returned, it interprets this to mean that you've fixed the problem and the faulting instruction should be restarted. Since my _except_handler function tweaked the EAX register to point to valid memory, the MOV EAX,1 instruction works the second time and function main continues normally. See, that wasn't so complicated, was it?

Moving In a Little Deeper

With this simplest of scenarios behind us, let's go back and fill in some of the blanks. While this exception callback is great, it's not a perfect solution. In an application of any size, it would be exceedingly messy to write a single function to handle exceptions that could occur anywhere in your application. A much more workable scenario would be to have multiple exception handling routines, each one customized to a particular section of your application. Wouldn't you know it, the operating system provides just this functionality.

Remember the EXCEPTION_REGISTRATION structure that the system uses to find the exception callback function? The first member of this structure, which I ignored earlier, is called prev. It's really a pointer to another EXCEPTION_REGISTRATION structure. This second EXCEPTION_REGISTRATION structure can have a completely different handler function. What's more, its prev field can point to a third EXCEPTION_REGISTRATION structure, and so on. Simply put, there's a linked list of EXCEPTION_REGISTRATION structures. The head of the list is always pointed to by the first DWORD in the thread information block (FS:[0] on Intel-based machines).

What does the operating system do with this linked list of EXCEPTION_REGISTRATION structures? When an exception occurs, the system walks the list of structures and looks for an EXCEPTION_REGISTRATION whose handler callback agrees to handle the exception. In the case of MYSEH.CPP, the handler callback agreed to handle the exception by returning the value ExceptionContinueExecution. The exception callback can also decline to handle the exception. In this case, the system moves on to the next EXCEPTION_REGISTRATION structure in the list and asks its exception callback if it wants to handle the exception. Figure 4 shows this process. Once the system finds a callback that handles the exception, it stops walking the linked list of EXCEPTION_REGISTRATIONs.

To show an example of an exception callback that doesn't handle an exception, check out MYSEH2.CPP in **Figure 5**. To keep the code simple, I cheated a bit and used a little compiler-level exception handling. Function main just sets up a _try/_except block. Inside the _try block is a call to the HomeGrownFrame function. This function is very similar to the code in the earlier MYSEH program. It creates an EXCEPTION_REGISTRATION record on the stack and points FS:[0] at it. After establishing the new handler, the function intentionally causes a fault by writing to a NULL pointer:

```
*(PDWORD)0 = 0;
```

The exception callback function, again called _except_ handler, is quite different than the earlier version. The code first prints out the exception code and flags from the ExceptionRecord structure that the function received as a pointer parameter. The reason for printing out the exception flags will become clear later. Since this _except_handler function has no intention of fixing the offending code, the function returns ExceptionContinueSearch. This causes the operating system to continue its search at the next EXCEPTION_REGISTRATION record in the linked list. For now, take my word for it; the next installed exception callback is for the _try/_except code in function main. The _except block simply prints out "Caught the exception in main()". In this case, handling the exception is as simple as ignoring that it happened.

A key point to bring up here is execution control. When a handler declines to handle an exception, it effectively declines to decide where control will eventually resume. The handler that accepts the exception is the one that decides where control will continue after all the exception handling code is finished. This has an important implication that's not immediately obvious.

When using structured exception handling, a function may exit in an abnormal manner if it has an exception handler that doesn't handle the exception. For instance, in MYSEH2 the minimal handler in the HomeGrownFrame function didn't handle the exception. Since somebody later in the chain handled the exception (function main), the printf after the faulting instruction never executes. In some ways, using structured exception handling is similar to using the setjmp and longjmp runtime library functions.

If you run MYSEH2, you'll find something surprising in the output. It seems that there's two calls to the _except_handler function. The first call is certainly understandable, given what you know so far. But why the second?

```
Home Grown handler: Exception Code: C0000005 Exception Flags 0
Home Grown handler: Exception Code: C0000027 Exception Flags 2
                                             EH_UNWINDING
Caught the Exception in main()
```

There's obviously a difference: compare the two lines that start with "Home Grown Handler." In particular, the exception flags are zero the first time though, and 2 the second time. This brings me to the subject of unwinding. To jump ahead a bit, when an exception callback declines to handle an exception, it gets called a second time. This callback doesn't happen immediately, though. It's a bit more complicated then that. I'll need to refine the exception scenario one final time.

When an exception occurs, the system walks the list of EXCEPTION_REGISTRATION structures until it finds a handler for the exception. Once a handler is found, the system walks the list again, up to the node that will handle the exception. During this second traversal, the system calls each handler function a second time. The key distinction is that in the second call, the value 2 is set in the exception flags. This value corresponds to EH_UNWINDING. (The definition for EH_UNWINDING is in EXCEPT.INC, which is in the Visual C++ runtime library sources, but nothing equivalent appears in the Win32 SDK.)

What does EH_UNWINDING mean? When an exception callback is invoked a second time (with the EH_UNWINDING flag), the operating system is giving the handler function an opportunity to do any cleanup it needs to do. What sort of cleanup? A perfect example is that of a C++ class destructor. When a function's exception handler declines to handle an exception, control typically doesn't exit from that function in a normal manner. Now, consider a function with a C++ class declared as a local variable. The C++ specification says that the destructor must be called. The second exception handler callback with the EH_UNWINDING flag is the opportunity for the function to do cleanup work such as invoking destructors and _finally blocks.

After an exception is handled and all the previous exception frames have been called to unwind, execution continues wherever the handling callback decides. Remember though that it's just not enough to set the instruction pointer to the desired code address and plunge ahead. The code where execution resumes expects that the stack and frame pointer (the ESP and EBP registers on Intel CPUs) are set to their values within the stack frame that handled the exception. Therefore, the handler that accepts a particular exception is responsible for setting the stack and frame pointers to values that they had in the stack frame that contains the SEH code that handled the exception.

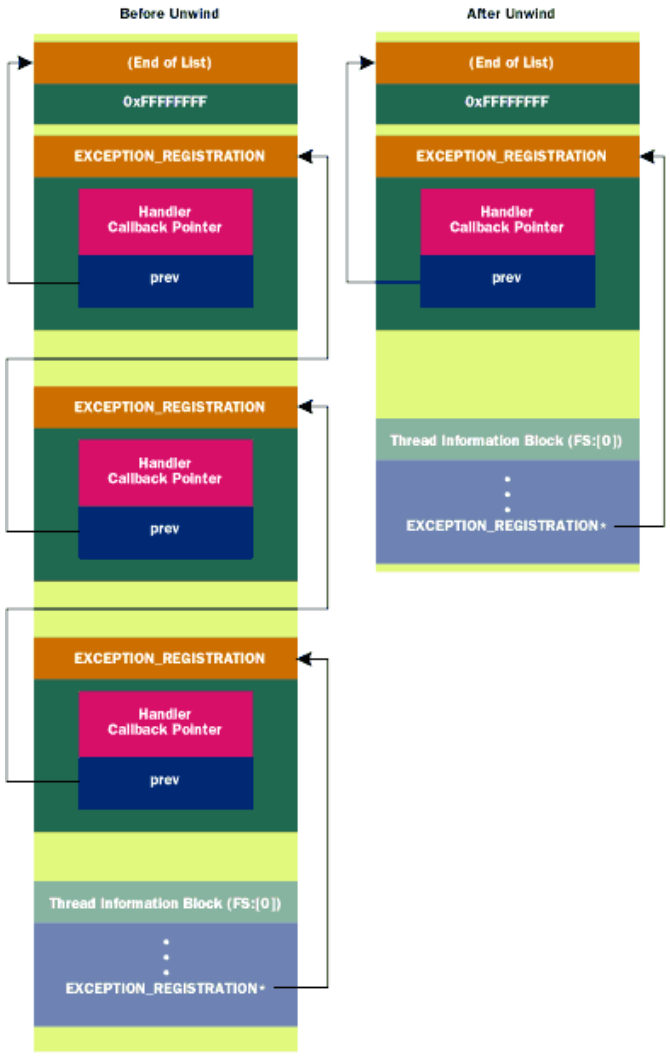

Figure 6 Unwinding from an Exception

In more general terms, the act of unwinding from an exception causes all things on the stack below the handling frame's stack region to be removed. It's almost as if those functions were never called. Another effect of unwinding is that all EXCEPTION_REGISTRATIONs in the list prior to the one that handled the exception are removed from the list. This makes sense, as these EXCEPTION_REGISTRATIONs are typically built on the stack. After the exception has been handled, the stack and frame pointers will be higher in memory than the EXCEPTION_REGISTRATIONs that were removed from the list. Figure 6 shows what I'm talking about.

Help! Nobody Handled It!

So far, I've implicitly assumed that the operating system always finds a handler somewhere in the linked list of EXCEPTION_REGISTRATIONs. What happens if nobody steps up to the plate? As it turns out, this almost never happens. The reason is that the operating system sneaks in a default exception handler for each and every thread. The default handler is always the

last node in the linked list, and it always chooses to handle the exception. Its actions are somewhat different than a normal exception callback routine, as I'll show later.

Let's look at where the system inserts the default, last resort exception handler. This obviously needs to occur very early in the thread's execution, before any user code executes. Figure 7 shows some pseudocode I wrote for BaseProcessStart, an internal routine in the Windows NT KERNEL32.DLL. BaseProcessStart takes one parameter, the address of the thread's entry point. BaseProcessStart runs in the context of the new process and calls the entry point to kick off execution of the first thread in the process.

In the pseudocode, notice that the call to lpfnEntryPoint is wrapped within an _try and _except construct. This _try block is what installs the default, last resort exception handler in the linked list of handlers. All subsequent registered exception handlers will be inserted ahead of this handler in the list. If the lpfnEntryPoint function returns, the thread ran to completion without causing an exception. When this happens, BaseProcessStart calls ExitThread to terminate the tread.

On the other hand, what if the thread faults and no other exception handlers handle it? In this case, control goes to the code inside the parens after the _except keyword. In BaseProcessStart, this code calls the UnhandledExceptionFilter API, which I'll come back to later. For now, the key point is that the UnhandledExceptionFilter API contains the meat of the default exception handler.

If UnhandledExceptionFilter returns EXCEPTION_EXECUTE_HANDLER, the _except block in BaseProcessStart executes. All the _except block code does is terminate the current process by calling ExitProcess. Thinking about it for a second, this makes sense; it's common knowledge that if a program causes a fault and nobody handles the fault, the system terminates the process. What you're seeing in the pseudocode is exactly where and how this happens.

There's one final addition to what I've just described. If the thread that faulted is running as a service and is for a thread-based service, the _except block code doesn't call ExitProcess—it calls ExitThread instead. You wouldn't want to terminate the entire service process just because one service went bad.

So what does the default exception handler code in UnhandledExceptionFilter do? When I ask this question at seminars, very few people can guess what the default behavior of the operating system is when an unhandled exception occurs. With a very simple demonstration of the default handler's behavior, things usually click and people understand. I simply run a program that intentionally causes a fault, and point out the results (see Figure 8).

Figure 8 Unhandled Exception Dialog

At a high level, UnhandledExceptionFilter displays a dialog telling you that a fault occurred. At that point, you're given the opportunity to either terminate or debug the faulting process. Much more happens behind the scenes, and I'll describe these things towards the end of the article.

As I've shown, when an exception occurs, user-written code can (and often does) get executed. Likewise, during an unwind operation, user-written code can execute. This user-written code may have bugs and could cause another exception. For this reason, there are two other values that an exception callback can return: ExceptionNestedException and ExceptionCollidedUnwind. While obviously important, this is pretty advanced stuff and I'm not going to dwell on it here. It's difficult enough to understand the basic facts.

Compiler-level SEH

While I've made occasional reference to _try and _except, just about everything I've written about so far is implemented by the operating system. However, in looking at the contortions of my two small programs that used raw system SEH, a compiler wrapper around this functionality is definitely in order. Let's now see how Visual C++ builds its structured exception handling support on top of the system-level SEH facilities.

Before moving on, it's important to remember that another compiler could do something completely different with the raw system-level SEH facilities. Nothing says that a compiler must implement the _try/_except model that the Win32 SDK documentation describes. For example, the upcoming Visual Basic® 5.0 uses structured exception handling in its runtime code, but the data structures and algorithms are completely different from what I'll describe here.

If you read through the Win32 SDK documentation on structured exception handling, you'll come across the following syntax for a so called "frame-based" exception handler:

```
try {
    // guarded body of code
}
except (filter-expression) {
    // exception-handler block
}
```

To be a bit simplistic, all of the code within a try block in a function is protected by an EXCEPTION_REGISTRATION that's built on the function's stack frame. On function entry, the new EXCEPTION_REGISTRATION is put at the head of the linked list of exception handlers. After the end of the _try block, its EXCEPTION_REGISTRATION is removed from the head of the list. As I mentioned earlier, the head of the exception handler chain is kept at FS:[0]. Thus, if you're stepping through assembly language in a debugger and you see instructions such as

```
MOV DWORD PTR FS:[00000000],ESP
```

or

```
MOV DWORD PTR FS:[00000000],ECX
```

you can be pretty sure that the code is setting up or tearing down a _try/_except block.

Now that you know that a _try block corresponds to an EXCEPTION_REGISTRATION structure on the stack, what about the callback function within the EXCEPTION_ REGISTRATION? Using Win32 terminology, the exception callback function corresponds to the filter-expression code. To refresh your memory, the filter-expression is the code in parens after the _except keyword. It's this filter-expression code that decides whether the code in the subsequent {} block will execute.

Since you write the filter-expression code, you get to decide if a particular exception will be handled at this particular point in your code. Your filter-expression code might be as simple as saying "EXCEPTION_EXECUTE_ HANDLER." Alternatively, the filter-expression might invoke a function that calculates p to 20 million places before returning a code telling the system what to do next. It's your choice. The key point: your filter-expression code is effectively the exception callback that I described earlier.

What I've just described, while reasonably simple, is nonetheless a rose-colored-glasses view of the world. The ugly reality is that things are more complicated. For starters, your filter-expression code isn't called directly by the operating system. Rather, the exception handler field in every EXCEPTION_REGISTRATION points to the same function. This function is in the Visual C++ runtime library and is called __except_handler3. It's __except_handler3 that calls your filter-expression code, and I'll come back to it a bit later.

Another twist to the simplistic view that I described earlier is that EXCEPTION_REGISTRATIONs aren't built and torn down every time a _try block is entered or exits. Instead, there's just one EXCEPTION_REGISTRATION created in any function that uses SEH. In other words, you can have multiple _try/_except constructs in a function, yet only one EXCEPTION_REGISTRATION is created on the stack. Likewise, you might have a _try block nested within another _try block in a function. Still, Visual C++ creates only one EXCEPTION_REGISTRATION.

If a single exception handler (that is, __except_handler3) suffices for the whole EXE or DLL, and if a single EXCEPTION_REGISTRATION handles multiple _try blocks, there's obviously more going on here than meets the eye. This magic is accomplished through data in tables that you don't normally see. However, since the whole purpose of this article is to dissect structured exception handling, let's take a look at these data structures.

The Extended Exception Handling Frame

The Visual C++ SEH implementation doesn't use the raw EXCEPTION_REGISTRATION structure. Instead, it adds additional data fields to the end of the structure. This additional data is critical to allowing a single function (__except_handler3) to handle all exceptions and route control to the appropriate filter-expressions and _except blocks throughout the code. A hint to the format of the Visual C++ extended EXCEPTION_REGISTRATION structure is found in the EXSUP.INC file from the Visual C++ runtime library sources. In this file, you'll find the following (commented out) definition:

```
;struct _EXCEPTION_REGISTRATION{
;     struct _EXCEPTION_REGISTRATION *prev;
;     void (*handler)(PEXCEPTION_RECORD,
;                     PEXCEPTION_REGISTRATION,
;                     PCONTEXT,
;                     PEXCEPTION_RECORD);
;     struct scopetable_entry *scopetable;
;     int trylevel;
```

```
;     int _ebp;
;     PEXCEPTION_POINTERS xpointers;
;};
```

You've seen the first two fields, prev and handler, before. They make up the basic EXCEPTION_REGISTRATION structure. What's new are the last three fields: scopetable, trylevel, and _ebp. The scopetable field points to an array of structures of type scopetable_entries, while the trylevel field is essentially an index into this array. The last field, _ebp, is the value of the stack frame pointer (EBP) before the EXCEPTION_REGISTRATION was created.

It's not coincidental that the _ebp field becomes part of the extended EXCEPTION_REGISTRATION structure. It's included in the structure via the PUSH EBP instruction that most functions begin with. This has the effect of making all of the other EXCEPTION_REGISTRATION fields accessible as negative displacements from the frame pointer. For example, the trylevel field is at [EBP-04], the scopetable pointer is at [EBP-08], and so on.

Immediately below its extended EXCEPTION_REGISTRATION structure, Visual C++ pushes two additional values. In the DWORD immediately below, it reserves space for a pointer to an EXCEPTION_POINTERS structure (a standard Win32 structure). This is the pointer returned when you call the GetExceptionInformation API. While the SDK documentation implies that GetException Information is a standard Win32 API, the truth is that GetExceptionInformation is a compiler-intrinsic function. When you call the function, Visual C++ generates the following:

```
MOV EAX,DWORD PTR [EBP-14]
```

Just as GetExceptionInformation is a compiler intrinsic function, so is the related function GetExceptionCode. GetExceptionCode just reaches in and returns the value of a field from one of the data structures that GetExceptionInformation returns. I'll leave it as an exercise for the reader to figure out exactly what's going on when Visual C++ generates the following instructions for GetExceptionCode:

```
MOV EAX,DWORD PTR [EBP-14]
MOV EAX,DWORD PTR [EAX]
MOV EAX,DWORD PTR [EAX]
```

Returning to the extended EXCEPTION_REGISTRATION structure, 8 bytes before the start of the structure, Visual C++ reserves a DWORD to hold the final stack pointer (ESP) after all the prologue code has executed. This DWORD is the normal value of the ESP register as the function executes (except of course when parameters are being pushed in preparation for calling another function).

If it seems like I've dumped a ton of information onto you, I have. Before moving on, let's pause for just a moment and review the standard exception frame that Visual C++ generates for a function that uses structured exception handling:

```
EBP-00 _ebp
EBP-04 trylevel
EBP-08 scopetable pointer
EBP-0C handler function address
EBP-10 previous EXCEPTION_REGISTRATION
EBP-14 GetExceptionPointers
EBP-18 Standard ESP in frame
```

From the operating system's point of view, the only fields that exist are the two fields that make up a raw EXCEPTION_REGISTRATION: the prev pointer at [EBP-10] and the handler function pointer at [EBP-0Ch]. Everything else in the frame is specific to the Visual C++ implementation. With this in mind, let's look at the Visual C++ runtime library routine that embodies compiler level SEH, __except_handler3.

__except_handler3 and the scopetable

While I'd dearly love to point you to the Visual C++ runtime library sources and have you check out the __except_handler3 function yourself, I can't. It's conspicuously absent. In its place, you'll have to make due with some pseudocode for __except_handler3 that I cobbled together (see Figure 9).

While __except_handler3 looks like a lot of code, remember that it's just an exception callback like I described at the beginning of this article. It takes the identical four parameters as my homegrown exception callbacks in MYSEH.EXE and MYSEH2.EXE. At the topmost level, __except_handler3 is split into two parts by an if statement. This is because the function can be called twice, once normally and once during the unwind phase. The larger portion of the function is devoted to the non-unwinding callback.

The beginning of this code first creates an EXCEPTION_POINTERS structure on the stack, initializing it with two of the __except_handler3 parameters. The address of this structure, which I've called exceptPtrs in the pseudocode, is placed at [EBP-14]. This initializes the pointer that the GetExceptionInformation and GetExceptionCode functions use.

Next, __except_handler3 retrieves the current trylevel from the EXCEPTION_REGISTRATION frame (at [EBP-04]). The trylevel variable acts as an index into the scopetable array, which allows a single EXCEPTION_REGISTRATION to be used for multiple _try blocks within a function, as well as nested _try blocks. Each scopetable entry looks like this:

```
typedef struct _SCOPETABLE
{
    DWORD       previousTryLevel;
    DWORD       lpfnFilter
    DWORD       lpfnHandler
} SCOPETABLE, *PSCOPETABLE;
```

The second and third parameters in a SCOPETABLE are easy to understand. They're the addresses of your filter-expression and the corresponding _except block code. The previous tryLevel field is bit trickier. In a nutshell, it's for nested try blocks. The important point here is that there's one SCOPETABLE entry for each _try block in a function.

As I mentioned earlier, the current trylevel specifies the scopetable array entry to be used. This, in turn, specifies the filter-expression and _except block addresses. Now, consider a scenario with a _try block nested within another _try block. If the inner _try block's filter-expression doesn't handle the exception, the outer _try block's filter-expression must get a crack at it. How does __except_handler3 know which SCOPETABLE entry corresponds to the outer _try block? Its index is given by the previousTryLevel field in a SCOPETABLE entry. Using this scheme, you can create arbitrarily nested _try blocks. The previousTryLevel field acts as a node in a linked list of possible exception handlers within the function. The end of the list is indicated by a trylevel of 0xFFFFFFFF.

Returning to the __except_handler3 code, after it retrieves the current trylevel the code points to the corresponding SCOPETABLE entry and calls the filter- expression code. If the filter-expression returns EXCEPTION_CONTINUE_SEARCH, __except_handler3 moves on to the next SCOPETABLE entry, which is specified in the previousTryLevel field. If no handler is found by traversing the list, __except_handler3 returns DISPOSITION_CONTINUE_SEARCH, which causes the system to move on to the next EXCEPTION_REGISTRATION frame.

If the filter-expression returns EXCEPTION_EXECUTE_HANDLER, it means that the exception should be handled by the corresponding _except block code. This means that any previous EXCEPTION_REGISTRATION frames have to be removed from the list and the _except block needs to be executed. The first of these chores is handled by calling __global_unwind2, which I'll describe later on. After some other intervening cleanup code that I'll ignore for the moment, execution leaves __except_handler3 and goes to the _except block. What's strange is that control never returns from the _except block, even though __except_handler3 makes a CALL to it.

How is the current trylevel set? This is handled implicitly by the compiler, which performs on-the-fly modifications of the trylevel field in the extended EXCEPTION_REGISTRATION structure. If you examine the assembler code generated for a function that uses SEH, you'll see code that modifies the current trylevel at [EBP-04] at different points in the function's code.

How does __except_handler3 make a CALL to the _except block code, yet control never returns? Since a CALL instruction pushes a return address onto the stack, you'd think this would mess up the stack. If you examine the generated code for an _except block, you'll find that the first thing it does is load the ESP register from the DWORD that's 8 bytes below the EXCEPTION_REGISTRATION structure. As part of its prologue code, the function saves the ESP value away so that an _except block can retrieve it later.

The ShowSEHFrames Program

If you're feeling a bit overwhelmed at this point by things like EXCEPTION_REGISTRATIONs, scopetables, trylevels, filter-expressions, and unwinding, so was I at first. The subject of compiler-level structured exception handling does not lend itself to learning incrementally. Much of it doesn't make sense unless you understand the whole ball of wax. When confronted with a lot of theory, my natural inclination is to write code that applies the concepts I'm learning. If the program works, I know that my understanding is (usually) correct.

Figure 10 is the source code for ShowSEHFrames.EXE. It uses _try/_except blocks to set up a list of several Visual C++ SEH frames. Afterwards, it displays information about each frame, as well as the scopetables that Visual C++ builds for each frame. The program doesn't generate or expect any exceptions. Rather, I included all the _try blocks to force Visual C++ to generate multiple EXCEPTION_ REGISTRATION frames, with multiple scopetable entries per frame.

The important functions in ShowSEHFrames are WalkSEHFrames and ShowSEHFrame. WalkSEHFrames first prints out the address of __except_handler3, the reason for which will be clear in a moment. Next, the function obtains a pointer to the head of the exception list from FS:[0] and walks each node in the list. Each node is of type VC_EXCEPTION_REGISTRATION, which is a structure that I defined to describe a Visual C++ exception handling frame. For each node in the list, WalkSEHFrames passes a pointer to the node to the ShowSEHFrame function.

ShowSEHFrame starts by printing the address of the exception frame, the handler callback address, the address of the previous exception frame, and a pointer to the scopetable. Next, for each scopetable entry, the code prints out the previous trylevel, the filter-expression address, and the _except block address. How do I know how many entries are in a scopetable? I don't really. Rather, I assume that the current trylevel in the VC_EXCEPTION_REGISTRATION structure is one less than the total number of scopetable entries.

Figure 11 shows the results of running ShowSEHFrames. First, examine every line that begins with "Frame:". Notice how each successive instance shows an exception frame that's at a higher address on the stack. Next, on the first three Frame: lines, notice that the Handler value is the same (004012A8). Looking at the beginning of the output, you'll see that this 004012A8 is none other than the address of the __except_handler3 function in the Visual C++ runtime library. This proves my earlier assertion that a single entry point handles all exceptions.

You may be wondering why there are three exception frames using __except_handler3 as their callback since ShowSEHFrames plainly has only two functions that use SEH. The third frame comes from the Visual C++ runtime library. The code in CRT0.C from the Visual C++ runtime library sources shows that the call to main or WinMain is wrapped in an _try/_except block. The filter-expression code for this _try block is found in the WINXFLTR.C file.

Returning to ShowSEHFrames, the Handler for the last Frame: line contains a different address, 77F3AB6C. Poking around a bit, you'll find that this address is in KERNEL32.DLL. This particular frame is installed by KERNEL32.DLL in the BaseProcessStart function that I described earlier.

Unwinding

Let's briefly recap what unwinding means before digging into the code that implements it. Earlier, I described how potential exception handlers are kept in a linked list, pointed to by the first DWORD of the thread information block (FS:[0]). Since the handler for a particular exception may not be at the head of the list, there needs to be an orderly way of removing all exception handlers in the list that are ahead of the handler that actually deals with the exception.

As you saw in the Visual C++ __except_handler3 function, unwinding is performed by the __global_unwind2 RTL function. This function is just a very thin wrapper around the undocumented RtlUnwind API:

```
__global_unwind2(void * pRegistFrame)
{
    _RtlUnwind( pRegistFrame,
                &__ret_label,
                0, 0 );
    __ret_label:
}
```

While RtlUnwind is a critical API for implementing compiler-level SEH, it's not documented anywhere. While technically a KERNEL32 function, the Windows NT KERNEL32.DLL forwards the call to NTDLL.DLL, which also has an RtlUnwind function. I was able to piece together some pseudocode for it, which appears in Figure 12.

While RtlUnwind looks imposing, it's not hard to understand if you methodically break it down. The API begins by retrieving the current top and bottom of the thread's stack from FS:[4] and FS:[8]. These values are important later as sanity checks to ensure that all of the exception frames being unwound fall within the stack region.

RtlUnwind next builds a dummy EXCEPTION_RECORD on the stack and sets the ExceptionCode field to STATUS_UNWIND. Also, the EXCEPTION_UNWINDING flag is set in the ExceptionFlags field of the EXCEPTION_RECORD. A pointer to this structure will later be passed as a parameter to each exception callback. Afterwards, the code calls the _RtlpCaptureContext function to create a dummy CONTEXT structure that also becomes a parameter for the unwind call of the exception callback.

The remainder of RtlUnwind traverses the linked list of EXCEPTION_REGISTRATION structures. For each frame, the code calls the RtlpExecuteHandlerForUnwind function, which I'll cover later. It's this function that calls the exception callback with the EXCEPTION_UNWINDING flag set. After each callback, the corresponding exception frame is removed by calling RtlpUnlinkHandler.

RtlUnwind stops unwinding frames when it gets to the frame with the address that was passed in as the first parameter. Interspersed with the code I've described is sanity-checking code to ensure that everything looks okay. If some sort of problem crops up, RtlUnwind raises an exception to indicate what the problem was, and this exception has the EXCEPTION_NONCONTINUABLE flag set. A process isn't allowed to continue execution when this flag is set, so it must terminate.

Unhandled Exceptions

Earlier in the article, I put off a full description of the UnhandledExceptionFilter API. You normally don't call this API directly (although you can). Most of the time, it's invoked by the filter-expression code for KERNEL32's default exception callback. I showed this earlier in the pseudocode for BaseProcessStart.

Figure 13 shows my pseudocode for UnhandledExceptionFilter. The API starts out a bit strangely (at least in my opinion). If the fault is an EXCEPTION_ACCESS_ VIOLATION, the code calls _BasepCheckForReadOnlyResource. While I haven't provided pseudocode for this function, I can summarize it. If the exception occurred because a resource section (.rsrc) of an EXE or DLL was written to, _BasepCurrentTopLevelFilter changes the faulting page's attributes from its normal read-only state, thereby allowing the write to occur. If this particular scenario occurs, UnhandledExceptionFilter returns EXCEPTION_ CONTINUE_EXECUTION and execution restarts at the faulting instruction.

The next task of UnhandledExceptionFilter is to determine if the process is being run under a Win32 debugger. That is, the process was created with the DEBUG_PROCESS or DEBUG_ONLY_THIS_PROCESS flag. UnhandledExceptionFilter uses the NtQueryInformationProcess Function that I describe in this month's Under the Hood column to tell if the process is being debugged. If so, the API returns EXCEPTION_CONTINUE_SEARCH, which tells some other part of the system to wake up the debugger process and tell it that an exception occurred in the debuggee.

Next on UnhandledExceptionFilter's plate is a call to the user-installed unhandled exception filter, if present. Normally, there isn't a user-installed callback, but one can be installed via the SetUnhandledExceptionFilter API. I've also provided pseudocode for this API. The API simply bashes a global variable with the new user callback address, and returns the value of the old callback.

With the preliminaries out of the way, UnhandledExceptionFilter can get down to its primary job: informing you of your ignominious programming blunder with the ever- stylish Application Error dialog. There are two ways that this dialog can be avoided. The first is if the process has called SetErrorMode and specified the SEM_NOGPFAULTERRORBOX flag. The other method is to have the Auto value under the AeDebug registry key set to 1. In this case, UnhandledExceptionFilter skips the Application Error dialog and automatically fires up whatever debugger is specified in the Debugger value of the AeDebug key. If you're familiar with "just in time debugging," this is where the operating system supports it. More on this later.

In most cases, neither of these dialog avoidance conditions are true and UnhandledExceptionFilter calls the NtRaiseHardError function in NTDLL.DLL. It's this function that brings up the Application Error dialog. This dialog waits for you to hit the OK button to terminate the process, or Cancel to debug it. (Maybe it's just me, but hitting Cancel to launch a debugger seems a little backward.)

If you hit OK in the Application Error dialog box, UnhandledExceptionFilter returns EXCEPTION_EXECUTE_HANDLER. The code that called UnhandledExceptionFilter usually responds by terminating itself (as you saw in the BaseProcessStart code). This brings up an interesting point. Most people assume that the system terminates a process with an unhandled exception. It's actually more correct to say that the system sets up things so that an unhandled exception causes the process to terminate itself.

The truly interesting code in UnhandledExceptionFilter executes if you select Cancel in the Application Error dialog, thereby bringing up a debugger on the faulting process. The code first calls CreateEvent to make an event that the debugger will signal after it has attached to the faulting process. This event handle, along with the current process ID, is passed to sprintf, which formats the command line used to start the debugger. Once everything is prepared, UnhandledExceptionFilter calls CreateProcess to start the debugger. If CreateProcess succeeds, the code calls NtWaitForSingleObject on the event created earlier. This call blocks until the debugger process signals the event, indicating that it has attached to the faulting process successfully. There are other little bits and pieces to the UnhandledExceptionFilter code, but I've covered the important highlights here.

Into the Inferno

If you've made it this far, it wouldn't be fair to finish without completing the entire circuit. I've shown how the operating system calls a user-defined function when an exception occurs. I've shown what typically goes on inside of those callbacks, and how compilers use them to implement _try and _catch. I've even shown what happens when nobody handles the exception and the system has to do the mopping up. All that remains is to show where the exception callbacks originate from in the first place. Yes, let's plunge into the bowels of the system and see the beginning stages of the structured exception handling sequence.

Figure 14 shows some pseudocode I whipped up for KiUserExceptionDispatcher and some related functions. KiUserExceptionDispatcher is in NTDLL.DLL and is where execution begins after an exception occurs. To be 100 percent accurate, what I just said isn't exactly true. For instance, in the Intel architecture an exception causes control to vector to a ring 0 (kernel mode) handler. The handler is defined by the interrupt descriptor table entry that corresponds to an exception. I'm going to skip all that kernel mode code and pretend that the CPU goes straight to KiUserExceptionDispatcher upon an exception

The heart of KiUserExceptionDispatcher is its call to RtlDispatchException. This kicks off the search for any registered exception handlers. If a handler handles the exception and continues execution, the call to RtlDispatchException never returns. If RtlDispatchException returns, there are two possible paths: either NtContinue is called, which lets the process continues, or another exception is raised. This time, the exception isn't continuable, and the process must terminate.

Moving on to the RtlDispatchExceptionCode, this is where you'll find the exception frame walking code that I've referred to throughout this article. The function grabs a pointer to the linked list of EXCEPTION_REGISTRATIONs and iterates over every node, looking for a handler. Because of the possibility of stack corruption, the routine is very paranoid. Before calling the handler specified in each EXCEPTION_REGISTRATION, the code ensures that the EXCEPTION_REGISTRATION is DWORD-aligned, within the thread's stack, and higher on the stack than the previous EXCEPTION_REGISTRATION.

RtlDispatchException doesn't directly call the address specified in the EXCEPTION_REGISTRATION structure. Instead, it calls RtlpExecuteHandlerForException to do the dirty work. Depending on what happens inside RtlpExecuteHandlerForException, RtlDispatchException either continues walking the exception frames or raises another exception. This secondary exception indicates that something went wrong inside the exception callback and that execution can't continue.

The code for RtlpExecuteHandlerForException is closely related to another function, RtlpExecutehandlerForUnwind. You may recall that I mentioned this function earlier when I described unwinding. Both of these "functions" simply load the EDX register with different values before sending control to the ExecuteHandler function. Put another way, RtlpExecuteHandlerForException and RtlpExecutehandlerForUnwind are separate front ends to a common function, ExecuteHandler.

ExecuteHandler is where the handler field from the EXCEPTION_REGISTRATION is extracted and called. Strange as it may seem, the call to the exception callback is itself wrapped by a structured exception handler. Using SEH within itself seems a bit funky but it makes sense if you ponder it for a moment. If an exception callback causes another exception, the operating system needs to know about it. Depending on whether the exception occurred during the initial callback or during the unwind callback, ExecuteHandler returns either DISPOSITION_NESTED_ EXCEPTION or DISPOSITION_COLLIDED_UNWIND. Both are basically "Red Alert! Shut everything down now!" kind of codes.

If you're like me, it's hard to keep all of the functions associated with SEH straight. Likewise, it's hard to remember who calls who. To help myself, I came up with the diagram shown in Figure 15.

Now, what's the deal with setting EDX before getting to the ExecuteHandler code? It's simple, really. ExecuteHandler uses whatever's in EDX as the raw exception handler if something goes wrong while calling the user-installed handler. It pushes the EDX register onto the stack as the handler field for a minimal EXCEPTION_REGISTRATION structure. In essence, ExecuteHandler uses raw structured exception handling like I showed in the MYSEH and MYSEH2 programs.

Conclusion

Structured exception handling is a wonderful Win32 feature. Thanks to the supporting layers that compilers like Visual C++ put on top of it, the average programmer can benefit from SEH with a relatively small investment in learning. However, at the operating system level, things are more complicated than the Win32 documentation would lead you to believe.

Unfortunately, not much has been written about system-level SEH to date because almost everyone considers it an extremely difficult subject. The lack of documentation on the system-level details hasn't helped. In this article, I've shown that system-level SEH revolves around a relatively simple callback. If you understand the nature of the callback, and then build additional layers of understanding on top of that, system-level structured exception handling really isn't so hard to grasp.

From the January 1997 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.