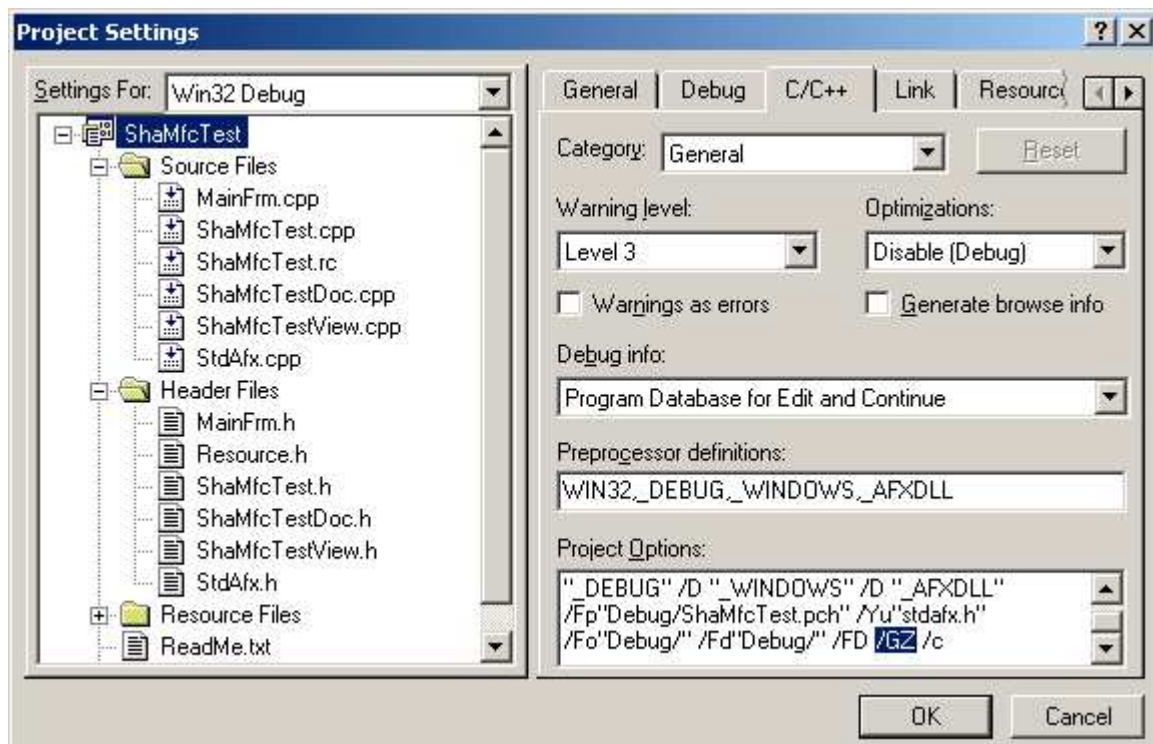


## The problem of MFC code decoration within RING0 execution code

When building MFC applications with SHA RING0 Execution, the code within the SHACODE section is not executed directly at application layer, rather it's mapped to the SHA SubSystem at system layer. The mapping of RING0 Execution code is done at runtime and requires non-decorated code by the compiler. Several application types (e.g. MFC DOC/VIEW application) have compiler settings which generate decorated code. For DEBUG builds in Microsoft® Visual C++® 6.0, the compiler switch /GZ is used by default. This switch performs the following:

- Auto-initialization of local variables
- Function pointer call stack validation
- Call stack validation



For function pointer call stack validation, the stack pointer (ESP) is checked to make sure it is the same before and after a call through a function pointer. This can catch any mismatch between the calling function's cleanup calling convention, and the called function's cleanup calling convention when called through a function pointer.

There are a few things to note about location:

- The first is that the routine in the runtime is memmove, which is perhaps not what you would expect. The Microsoft Visual C++ compiler is very smart when it comes to code generation and will take the easiest route. While writing this article, I called memset to set 20 bytes to a value. Rather than have the overhead of a loop, the compiler did five long movs.
- The second thing to note is that the parameters are quite clear—0x4d2 is our old friend 1234. However, ccccccc is less familiar. That is the uninitialized pointer *q*—it has taken on that value because the memory that represents the pointer was uninitialized. We can clearly see that the values being passed into the runtime are invalid in this case, and it is worth looking at other patterns that you might see while debugging:

Pattern	Description
0xFDFDFDFD	No man's land (normally outside of a process)
0xDDDDDDDD	Freed memory
0xCDCDCDCD	Uninitialized (global)
0xCCCCCCCC	Uninitialized locals (on the stack)

These values are undocumented and subject to change, but any sort of a signpost can be helpful while debugging. Just because you don't see those values doesn't mean that the values in memory are valid, of course. These are just some of the common patterns.

SHACODE sections can give rise to some interesting errors. Consider the following code fragment:

```
#pragma code_seg("SHACODE")

    void static RealtimeTask(void)
    {
        SHA_SYSTEM_READPORT_UCHAR(0x3f8, &Value);
    }

#pragma code_seg()
```

If we look at the assembler for this, it is quite characteristic:

```
push     ebp
mov      ebp,esp
sub      esp,48h
push     ebx
push     esi
push     edi

lea      edi,[ebp-48h]
mov      ecx,12h
mov      eax,0CCCCCCCCh
rep stos dword ptr [edi]

mov      dword ptr [pValue],872FA008h
mov      word ptr [Address],offset RealtimeTask+23h (00416023)
mov      dx,word ptr [Address]
in       al,dx
mov      ecx,dword ptr [pValue]
mov      byte ptr [ecx],al
pop      edi
pop      esi
pop      ebx

add      esp,48h
cmp      ebp,esp
call    _chkesp (00402ebe)

mov      esp,ebp
pop      ebp
ret
```

This code above would result in a dramatically error , since the code mapping is not able to follow the decoration. Quite often the calculated address will be outside of the array but within the process space. In these cases, an access violation results straight to a system crash. To run the code properly in RINGO execution the option /GZ must be disabled. So we can easily see the difference in code:

```
push    ebp
mov     ebp,esp
sub     esp,48h
push    ebx
push    esi
push    edi
mov     dword ptr [pValue],87176008h
mov     word ptr [Address],offset RealtimeTask+14h (00415014)
mov     dx,word ptr [Address]
in     al,dx
mov     ecx,dword ptr [pValue]
mov     byte ptr [ecx],al
pop     edi
pop     esi
pop     ebx
mov     esp,ebp
pop     ebp
ret
```