# Why does the x64 calling convention reserve four home spaces for parameters even for functions that take fewer than four parameters?

devblogs.microsoft.com/oldnewthing/20160623-00

June 23, 2016

Raymond Chen

In the comments to Can an x64 function repurpose parameter home space as general scratch space?, many people questioned why the full four home spaces are allocated for all functions, even those that have fewer that four parameters.

MSDN gives a very brief answer: "This aids in the simplicity of supporting C unprototyped functions, and vararg C/C++ functions."

Let's dig into that sentence.

Classic C allows you to call an unprototyped function. You can just say "There's a function called `f`, and it returns an integer, but that's all I'm going to tell you. You'll just have to trust me on the rest."

```
int f();
```

You can call this function and pass however many parameters you like. As long as you pass enough parameters to satisfy the function, that's good enough. (Modern C has tightened the rules, but classic C didn't care.) The catch is that the number of parameters you pass could be *fewer* than the number of parameters the function actually accepts.

Huh?

The classic example of this is the `open` function. It takes three parameters, but the third parameter is required only when creating a file. If you aren't creating a file, then the third parameter can be omitted.

```
fd = open(filename, O_RDONLY);
fd = open(filename, O_CREAT | O_WRONLY,  0777);
```

Now think about how this function would be implemented. On entry, the `rcx` register points to the file name, the `rdx` register contains the flags, and the `r8` register *might or might not* contain the creation mode.

So how do you know whether you should spill `r8`? If you spill it when you shouldn't have, then you corrupted the stack.

Okay, so you can work around this by spilling to the local frame instead of onto the home space, but since the compiler doesn't know whether this function is ever going to be called with fewer parameters than prototyped, it means that the compiler effectively *can never use the home space in the first place*, because, who knows, maybe the function goes like this:

```
int open(char *filename, int flags, int mode)
{
  if (is_auto_fail_mode()) return -1;
  ... rest of the code ...
}
```

and the caller cleverly arranged that `is_auto_fail_mode()` will return `1`, and then called `open()` and intentionally passed no parameters at all. Under classic C, this is perfectly legal.

If you say that the caller needs to allocate home space only for the actual parameters, then the result is that the compiler can never rely on the existence of home space. Which kind of renders home space useless.

Variadic functions are in a similar boat. If home space was guaranteed only for actual parameters, then variadic functions would not know whether any of the inbound register parameters are spillable. They would have to spill them into the local frame, but that makes walking the variable parameter list more cumbersome because the parameters are no longer contiguous in memory. You have the four locally-spilled parameters, followed by the function exception state, the frame pointer, the return address, and then parameters beyond the fourth.

I guess you could hack around this by changing the prologue of variadic functions to go something like this:

```
    sub     rsp, N+32       ;
    mov     rax, [rsp+N+32] ; recover return address
    mov     [rsp+N+32], r9  ; spill parameter 4 (if it exists)
    mov     [rsp+N+24], r8  ; spill parameter 3 (if it exists)
    mov     [rsp+N+16], rdx ; spill parameter 2 (if it exists)
    mov     [rsp+N+ 8], rcx ; spill parameter 1 (if it exists)
    mov     [rsp+N   ], rax ; restore return address
    .. rest of standard prologue to build the frame and stuff ..
```

and the function epilogue would go

```
    add     rsp, N
    ret     32
```

to clean up the four parameters that got rewritten on the stack.

This would fall into the “pay for play” category, where the ugliness is localized only to the people who need the ugly thing. It would make stack walking harder, but stack walking is relatively infrequent, so making stack-walking harder isn’t that big of a deal; as long as stack-walking isn’t rendered *impossible*.

But wait, why is reserving the full four home spaces so distasteful in the first place? Maybe it’s because people consider it wasteful to allocate memory that isn’t being used.

But is it really going unused?

No, it’s not going unused. The whole point of the original article was to say that the four spaces for home parameters aren’t actually required to be used for home parameter spilling. A function is welcome to treat it as *hey look, free memory*. and in practice, that’s what most of them do. They will spill `rbx` and `rsi` into those spaces rather than spilling the actual parameters. So the memory isn’t going to waste.

Requiring the full four home parameters to be preallocated also saves you the trouble of having to keep realigning your stack on a 16-byte boundary for each function call. The value 32 is a multiple of 16, so whatever alignment you already performed is unaffected by subtracting another 32. On the other hand, if you had callee clean and variable-sized home space, you would have to push an extra dummy parameter for half of the functions, just so that the stack stayed aligned.

```
    ; about to call a function that takes only 1 parameter.
    ; need to push a dummy parameter to keep the stack 16-byte aligned.
    push    0
    push    rax
    call    f
```

But you can solve that problem by saying that functions are always caller-clean. That way, you just set up your register spill space once, and then you keep reusing it. It also has the nice property that local variables remain at the same offset relative to the stack pointer for the lifetime of the function.

Or maybe the objection is that it creates more work for the caller, having to allocate 32 extra bytes of memory.

Except that in practice, it’s not any extra work at all. As we noted, the x64 calling convention is caller-clean, which means that the space for parameters gets reused from function call to function call. You merely allocate the space in your prologue and it’s good for the whole function. And you already had to reserve space on the stack in the function prologue when you did a `sub esp, N`; you just need to bump it up to `sub esp, N+32`. This is arithmetic done at compile time, so there is no additional runtime cost. (It’s not like the CPU goes faster if the subtrahend is smaller.)

The upshot of this is that trying to minimize parameter home space doesn't really save you anything. Memory isn't being saved, because the called function was going to use the extra space anyway. Execution time isn't being saved, because the stack pointer adjustment is already being done; you're just changing the amount of the adjustment, which has no incremental cost. You're creating more work for the compiler, because it now needs to keep track of the maximum number of parameters passed to any function called by this function. (Mind you, that's not a big cost anyway. Itanium relied on the compiler doing this work.) And you're creating more work for the function being called, because it has to be careful not to spill any registers which *might not* correspond to actual parameters (information that is not available at compile time).

So reserving home space for all four parameters (even if the caller passes fewer than four actual parameters) makes some people slightly happier (variadic functions), a lot of people significantly happier (classic C code), and adds zero additional cost. Free money.

Raymond Chen

**Follow**