

## Re: Unions in Assembly Language

**Source:** <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2004-08/1493.html>

---

**From:** The Wannabee (*faq\_at\_@.SZMyggenPV.com*)

**Date:** 08/26/04

Date: Thu, 26 Aug 2004 17:56:13 +0200

På Thu, 26 Aug 2004 16:59:17 +0200, skrev f0dder  
<f0dder\_spicedham@flork.dk>:

> *The Wannabee wrote:*

>

>> *Yes. Jump table or callable is fine to me. I made a typo in the  
>> original post. First I asked Percival, because he is a HLA  
>> programmer, because I wanted to see the HLA equivalent. But to see  
>> more implementations is nice too. Since Percival didnt adress the  
>> full query, here is the RosAsm equivalent of you C code : (btw, your  
>> c code doesnt include dessionmaking for the table, so I will also  
>> not include this, although I had expected this to be part of the  
>> solution, as otherwise the solution will be less useful)*

>>

>>

> *decision making? Do you mean bounds checking?*

No I mean, one of the purposes of a callable, or a LOOKUP/CALLTABLE is to speed up redirection of code, that would otherwise, because the choices of redirection grew, would make simple repeated cmp / call code be somewhat slower. According to wolfgang, as soon as a if(then/else (cmp/jcc) statement approaches 5-8 selections, a callable or jumtable will execute faster, and when the dessionmaking is very large, the speed of the lookup will not increase (except when cache is affected), it will be constant.

In your example, the code didnt need a callable, as it only used the callable to call the procs indirectly, while for the example it would be better to make the call directly. However, it did show how to prepare the tables, and how to call them. Only the dession making and the automated calling was omitted. What I mean by this is for example :

```
[WinMessagesJumpTable:  
  0 0 ;1  
  Application.WMDestroy ;2  
  0 0 ;4  
  Application.WMSize ;5  
  0 0 0 0 0 0 0 ;14
```

```
Application.WMPaint ;15
0 0 0 0 ;19
Application.WMEraseBkGnd ;20
.....
]
```

and the code : (This will crash if EDI is set to NULL on exit (Win98 1st edition),

but works if EDI points to application local memory, understand it those who can)

```
[mRet:0]
Application.Dispatch:
    pop D$mret
    cmp D$esp+4 MaxMessageTable|jae @NoHandler
    mov eax D$esp + 0_4
    mov ecx D$WinMessagesJumpTable + eax*4
    cmp ecx 0 |je @NoHandler

    mov eax D$esp + 0_4
    mov ecx D$WinMessagesJumpTable + eax*4
    call ecx
```

or eax eax ; Means message was handel by app \_OR\_ need not defaulthandling

```
    jne @MessageWasHandled
```

@NoHandler:

```
    call 'User32.DefWindowProcA'
```

@MessageWasHandled:

```
    jmp D$mret
```

>> *The RosAsm equivalent of your code :*

>>

> *\*snip\**

> *What VC++ generates is not too far from your asm code – except it*

> *uses normal stack calling convention.*

Yes. This is a another detail I picked up from WolfGang as well, since the stack is memory, it can break optimal cache usage, because if some memory is allready cached, and you then push or pop, that memory which was cached, can/will be trashed. I dont understand every detail of it, but for fast code, stack should be avoided when not absolutely needed. Same for calls, where the values are pushed popped needlessly into and off the stack, and could potentially trash data, which may allready be in the cache. I am far from as clever as he on these details, but anyway, it is useful info that could come in handy in the right situation. I am not the person who needs to have absolute best code/speed everywhere, but like to include tips picked up into newly written code. I assume for instance that if you have used some code to load the caches with some data, and then make a call, using the stack, may force the data to be reread from main memory on the first access in the target procedure. So avoiding the stack

would maybe prevent it.

But it depends assumefully on the code in question. Buy many times I guess in a more complex procedure that calls to subroutines to manipulate the same data as in the calling procedure, maybe during a loop, avoiding stack usage might make every call a bit faster. Anyway, I am way to inexperienced to give a full insight into these topics.

My point is that theres not anything significant that can be done in HLL that cannot be done faster, better and more optimal in asm, once the design is settled. And then the implementation can be bettered by each glance on the code in assembly, but in the HLL, because of the hidden details, it can look optimal, and not be so. And to learn the full potential of asm, I am convienced one must use asm daily, for years to see the full range of benfits. Better start now.

However, to understand that HLA is not assembly takes but a few weeks.