How to refer to the start-of a user-defined segment in a Visual Studio-project?

I'm struggling to convert a C-program linked with ld, of the gnu tool-chain to make it compile as a visual-studio (2005) project. The program puts .data-symbols in different segments and during an initialization phase it copies data between segments. Pointers to the start and end of the segments are defined in the ld linker script.

I understand how to locate the variables into different, user-defined segments, but i havent been able to figure out how to define linker constants such as _start_of_my_segment or if there is something similar to a linker script in Visual Studio.

My goal is to be able to compile the program with, prefferably no modifications to the source-code that refers to the linker-defined symbols, but with my own custom layout of the data in the Visual Studio project.

Below is some example C-code that illustrates what i'd like to do and a (stripped-down, possibly syntax-incorrect) version of the make-script used when linking with gcc/ld.

Any hints would be greatly appreciated!

```
#pragma data_seg( "MY_DATA_FOO" )
#pragma data_seg( "MY_DATA_BAR" )
#pragma comment(linker, "/section:MY_DATA_BAR,R")

__declspec(allocate("MY_DATA_FOO")) int foo1;
__declspec(allocate("MY_DATA_FOO")) int foo2;

__declspec(allocate("MY_DATA_BAR")) int bar1 = 1;
__declspec(allocate("MY_DATA_BAR")) int bar2 = 2;

#pragma data_seg( )
void test() {
    foo1 = bar1;
    foo2 = bar2;

    // i would rather do this as
    //extern unsigned int __start_of_MY_DATA_FOO;
    //extern unsigned int __start_of_MY_DATA_BAR;
    //extern unsigned int __size_of_MY_DATA_BAR;
    //memcpy(__start_of_MY_DATA_FOO, _start_of_MY_DATA_BAR, _size_of_MY_DATA_BAR);
}
```

Pseudo link-script (what would be the equivalent for Visual Studio

```
MEMORY
{
  foo:  org=0x1000, len=0x100
  bar:  org=0x2000, len=0x100
}

SECTIONS
{
    GROUP:
    {
        MY_DATA_FOO : {}
        __start_of_MY_DATA_FOO = ADDR(MY_DATA_FOO);
        __end_of_MY_DATA_FOO = .;
        __size_of_MY_DATA_FOO = SIZEOF(MY_DATA_FOO);
    } > foo

    GROUP:
    {
        MY_DATA_BAR : {}
        __start_of_MY_DATA_BAR = ADDR(MY_DATA_BAR);
        __end_of_MY_DATA_BAR = .;
        __size_of_MY_DATA_BAR = SIZEOF(MY_DATA_BAR);
    } > bar
}
```

c visual-c++ linker symbols segment

asked Jul 7 '10 at 7:59

ara
146 2 6

This is really something that should not be done. Surely there are portable ways to write the same program. Sounds like someone thought they were clever fooling with low-level build chain internals instead of using C properly... It's about one step above assuming you can access local variables from a called function in its caller after it returns... – R.. Jul 7 '10 at 10:42

@ R.: There are rare but occasionally decent reasons to use this pattern. I've used it for adding debug checks. In release builds, my objects are independent and don't know about each other. But for debugging, they do. The "legal C/C++" way is a central register to track this, made only for debugging, is actually a lot more maintenance for 200+ things than this approach is (this approach is automatic). On the other hand, importability is a different cost which this brings. – VoidStar Jul 28 '15 at 9:53

2 Answers

padding can be removed by segments merging

for example

```
#pragma data_seg(".foo_begin")
#pragma data_seg(".foo_data")
#pragma data_seg(".foo_end")

#pragma comment( linker, "/merge:.foo_begin=.foo" )
#pragma comment( linker, "/merge:.foo_data=.foo" )
#pragma comment( linker, "/merge:.foo_end=.foo" )

__declspec(allocate(".foo_begin")) int foo_begin_marker;
__declspec(allocate(".foo_end")) int foo_end_marker;

__declspec(allocate(".foo_data")) int foo_data;
```

answered Aug 6 '11 at 21:35

dnk
31 2

1 Thanks! However, I still get 0x100 bytes padding between the variables, (don't you?). Without the merge-command the padding is 0x1000 bytes and the segments end up in the wrong order, so the merge-command does have some effect and seems equally useful as the other approach (using $ in section names). – ara Aug 15 '11 at 15:54

Create additional segments (they are placed in memory alphabetically):

```
#pragma data_seg("MY_DATA_FOO__a")
#pragma data_seg("MY_DATA_FOO__z")
#pragma data_seg("MY_DATA_FOO__m")

__declspec(allocate("MY_DATA_FOO__a")) int fooFirst;
__declspec(allocate("MY_DATA_FOO__z")) int fooLast;
__declspec(allocate("MY_DATA_FOO__m")) int foo1;
__declspec(allocate("MY_DATA_FOO__m")) int foo2;
```

Then copy everything between &fooFirst and &fooLast.

answered Jul 7 '10 at 8:40

Sergey Podobry
4,853 16 33

Thanks for the suggestion. However, when i try it, the address of fooFirst is NOT less than fooLast, and the &foo1 is greater than &fooLast. Also there seems to be some padding to align the segments to even 0x1000 blocks. This might be adjustable by some compiler- and linker-settings. Any hints on what settings you used? – ara Jul 7 '10 at 13:51

Thanks again! I found an example of the trick you suggest in the doc for #pragma init_seg (here msdn.microsoft.com/en-us/library/7977wcck.aspx). It says "Section names must be 8 characters or less. The sections with the same name before the $ are merged into one section. The order that they are merged is determined by sorting the characters after the $." And, when I changed the section-names to "MY_DATA_FOO$a" etc they ended up correctly sorted. However each section-size's is still zero-padded up to approx 0x100 bytes which makes it hard to know the "proper" end of the segment. – ara Jul 7 '10 at 17:35

Here is an example from ATL sources: #pragma section("ATL$__a", read, shared) #pragma section("ATL$__z", read, shared) #pragma section("ATL$__m", read, shared) Try section instead of data_seg. – Sergey Podobry Jul 7 '10 at 18:21

Yes, they both work (if one uses '$' in the section name). However both still add some padding so i still cant figure out how to construct the symbol __size_of_MY_DATA_BAR. – ara Jul 7 '10 at 19:03

Hmmm... I've got an alignment too. And it is quite strange. – Sergey Podobry Jul 9 '10 at 6:31