说明: 本文搬运自我的个人博客,原地址点击打开链接

一. 关键字和运算符

1. __align(*n*)： 指示编译器在 *n* 字节边界上对齐**变量**。

对于局部变量，*n* 值可为 1、2、4 或 8。

对于全局变量，*n* 可以具有最大为 2 的 0x80000000 次幂的任何值。

__align 关键字紧靠变量名称**前面**放置。

注意事项：

只能进行过对齐。也就是说，可以将两个字节的对象按 4 个字节对齐，而不能将 4 个字节的对象按两个字节对齐。

用法举例：

[cpp] view plaincopy

```cpp
__align(8) char buffer[128];    /* buffer 从 8 字节对齐边界开始*/

void foo(void)
{
    ...
    __align(16) int i;          /* 这个对齐值是不允许用在局部变量中的*/
    ...
}

__align(16) int i;              /* 作为一个全局变量,这个对齐值是被允许的*/
```

2. __asm： 此关键字用于将信息从编译器传递到 ARM 汇编器 armasm。

用法：

a. 嵌入式汇编器

可以使用 __asm 关键字声明或定义嵌入式汇编函数。例如：

[cpp] view plaincopy

```cpp
__asm void my_strcpy(const char *src, char *dst);
```

b. 内联汇编器

可以使用 __asm 关键字将内联汇编合并到函数中。例如：

[cpp] view plaincopy

```cpp
int qadd(int i, int j)
{
    int res;
    __asm
    {
        QADD   res, i, j
    }
    return res;
}
```

c. 汇编器标签

可以使用 __asm 关键字为 C 符号指定汇编器标签。例如：

[cpp] view plaincopy

```cpp
int count __asm__("count_v1"); // export count_v1, not count
```

d. 已命名的寄存器变量

可以使用 __asm 关键字声明已命名的寄存器变量。例如：

[cpp] view plaincopy

```cpp
register int foo __asm("r0");
```

3. __forceinline：强制编译器内联编译 C 或 C++ 函。

说明：

__forceinline 的语义与 C++ inline 关键字的语义完全相同。编译器尝试内联限定为 __forceinline 的函数，而不考虑其特性。但是，如果这样做导致出现问题，编译器将不内联函数。例如，递归函数仅内联到本身一次。

注：

此关键字具有等效的函数属性 __attribute__((always_inline))。

用法举例：

[cpp] view plaincopy

```cpp
__forceinline static int max(int x, in y)
{
    return x > y ? x : y; // always inline if possible
}
```

4.__inline：提示编译器在**合理的情况下**内联编译 C 或 C++ 函数。

说明：

__inline 的语义与 C++ inline 关键字的语义完全相同。

用法举例：

[cpp] view plaincopy

```cpp
__inline int f(int x)
{
    return x*5+1;
}
int g(int x, int y)
{
    return f(x) + f(y);
}
```

5. __packed：将所有有效类型的对齐边界设置为 1。这就意味着：

- **不会插入填充以对齐压缩对象**
- **使用未对齐的访问读取或写入压缩类型的对象。**

使用 __packed 限定符声明结构或联合后，__packed 将应用于该结构或联合的所有成员。成员之间或结构末尾均没有填充。必须使用__packed 声明压缩结构的所有子结构。可以单独压缩非压缩结构的整型子字段。

用法：

若要将结构映射到外部数据结构或访问未对齐数据，__packed 限定符非常有用；但由于访问开销相对较高，通常对节省数据大小并没有什么帮助。通过仅对需要压缩的结构中的字段进行压缩，可以减少未对齐访问的数量。

注：

在硬件中不支持未对齐访问的 ARM 处理器（例如，ARMv6 之前的处理器）上，访问未对齐的数据时可能会在代码大小和执行速度方面产生较高的成本。必须**最大限度减少**通过压缩结构进行的数据访问，以避免增加代码大小和降低性能。

用法举例 1 – 压缩结构

[cpp] view plaincopy

```cpp
typedef __packed struct
{
    char x;                     // 结构体所有成员都会被__packed 限定
    int y;
} X;                            // 结构体占 5 字节，如果不使用__packed，这个结构占 8 字节
int f(X *p)
{
    return p->y;                // 执行一次未对齐读数据
}
typedef struct
{
    short x;
    char y;
    __packed int z;             // 仅这一个变量受到__packed 约束
    char a;
} Y;                            // 这个结构体占 8 字节
int g(Y *p)
{
    return p->z + p->x;         // 只有读区域 z 是未对齐的
}
```

用法举例 2 - 指向压缩类型的指针

[cpp] view plaincopy

```cpp
typedef __packed int* PpI;          /* 指向一个__packed int 型变量*/
__packed int *p;                    /* 指向一个__packed int 型变量*/
PpI p2;                             /* 'p2' 和'p'的类型相同 */
                                    /* __packed 可看作一个限定符*/
                                    /* 就像'const'或者'volatile'一样 */
typedef int *PI;                    /* 指向一个 int 型变量 */
__packed PI p3;                     /* 一个指向正常 int 型变量的__packed 指针*/
                                    /* -- 'p'和'p2'不是一个类型 */
int *__packed p4;                   /* 'p4'和'p3'的类型相同 */
```

二. 函数属性

1.__attribute__((always_inline))：此函数属性指示必须内联函数。

同__forceinline。

用法举例：

[cpp] view plaincopy

```cpp
<strong>   </strong> static int max(int x, int y) __attribute__((always_inline))
    {
        return x > y ? x : y; // always inline if possible
    }
```

2.__attribute__((used))：指示编译器在对象文件中保留静态函数，即使将该函数解除引用也是如此。

用法举例：

```
static int keep_this(int) __attribute__((used));      /*保留目标文件，编译器不进行空间优化*/
```

3.__attribute__((unused)):unused 函数属性禁止编译器在未引用该函数时生成警告。这不会更改删除未使用函数的过程的行为。

用法举例:

```
  static int Function_Attributes_unused_0(int b) __attribute__
((unused));
```

4.__attribute__((section("*name*"))):可以使用 section 函数属性将代码放在映像的不同节中。

用法举例:

在以下示例中，将 Function_Attributes_section_0 放在 RO 节 new_section 中，而不是放在 .text 中。

```
1: void Function_Attributes_section_0 (void)
2:     __attribute__ ((section ("new_section")));
3: void Function_Attributes_section_0 (void)
4: {
5:     static int aStatic =0;
6:     aStatic++;
7: }
```

三. 变量属性

1.__attribute__((at(*address*)))：可以使用此变量属性指定变量的绝对地址。

变量放在其自己的节中，编译器将为包含变量的节指定适当的类型：

- 只读变量放在 RO 类型的节中。
- 已初始化的读写变量放在 RW 类型的节中。

  特别地，显式初始化为零的变量放在 RW 中，而不是放在 ZI 中。此类变量不适合编译器的 ZI 到 RW 优化。

- 未初始化的变量放在 ZI 类型的节中。

语法：

__attribute__((at(address)))

其中，address 是所需的变量地址。

注：

链接器并非始终能够放置 at 变量属性生成的节。如果无法将节放置在指定地址，链接器将显示一条错误消息。

用法举例：

[cpp] view plaincopy

```cpp
const int x1 __attribute__((at(0x10000))) = 10; /* RO */
int x2 __attribute__((at(0x12000))) = 10;       /* RW */
int x3 __attribute__((at(0x14000))) = 0;        /* RW, not ZI */
int x4 __attribute__((at(0x16000)));            /* ZI */
```

扩展：

在 Keil MDK 提供的 absacc.h 中，这样定义了宏__at：

[cpp] view plaincopy

```cpp
#ifndef __at
#define __at(_addr) __attribute__ ((at(_addr)))
#endif
```

所以只要在模块中包含 absacc.h 头文件，就可以使用__at 来指定变量的绝对地址了。

2. __attribute__((zero_init))：编译器不对修饰的变量进行零初始化

section 属性指定变量必须放在特定数据节中。zero_init 属性指定将没有初始值设定项的变量放在 ZI 数据节中。如果程序指定了初始值设定项，则会报告错误。

用法举例：

```
__attribute__((zero_init)) int x;                    /* in
section ".bss" */
```

[cpp] view plaincopy

```
1.  __attribute__((section("mybss"), zero_init)) int y;  /* in section "m
    ybss" */
```

3.__attribute__((section("name"))):通常，ARM 编译器将它生成的对象放在节中，如 data 和 bss。但是，您可能需要使用其他数据节，或者希望变量出现在特殊节中，例如，便于映射到特殊硬件。section 属性指定变量必须放在特定数据节中。如果使用 section 属性，则将只读变量放在 RO 数据节中，而将读写变量放在 RW 数据节中，除非您使用 zero_init 属性。在这种情况下，变量被放在 ZI 节中。

用法举例:

```
1: /* in RO section */
2: const int descriptor[3] __attribute__ ((section ("descr")))
= { 1,2,3 };
3: /* in RW section */
4: long long rw[10] __attribute__ ((section ("RW")));
5: /* in ZI section */

6: long long altstack[10] __attribute__ ((section ("STACK"),
zero_init));
```

4.__attribute__((used)):此变量属性指示编译器在对象文件中保留某个静态变量，即使解除了对该变量的引用也是如此。

用法举例:

```
1: static int lose_this = 1;
2: static int keep_this __attribute__((used)) = 2;     //
retained in object file
3: static int keep_this_too __attribute__((used)) = 3; //
retained in object file
```

5.__attribute__((unused)):通常，如果声明了某个变量，但从未对其进行引用，编译器将发出警告。此属性指示编译器您预计不会使用某个变量，并指示它在未使用该变量时不要发出警告。

用法举例：

```
1: void Variable_Attributes_unused_0()
2: {
3:     static int aStatic =0;
4:     int aUnused __attribute__ ((unused));
5:     int bUnused;
6:     aStatic++;
7: }
```

四. 内联指令函数

1.__breakpoint：此内在函数在编译器生成的指令流中插入 BKPT 指令。它允许在 C 或 C++ 代码中包含断点指令。

语法：

```
void __breakpoint(int val)
```

其中，*val* 是编译时常数整数，其范围是：

0 . . . 65535

如果要将源代码编译为 ARM 代码

0 . . . 255

如果要将源代码编译为 Thumb 代码。

注：

为不支持 BKPT 指令的目标进行编译时，编译器无法识别 __breakpoint 内在函数。在这种情况下，编译器将生成警告或错误。

如果在不支持 BKPT 指令的体系结构上执行该指令，则会生成未定义的指令陷阱。

用法举例：

```
[cpp] view plaincopy
```

```
1.  void func(void)
2.  {
3.      ...
4.       __breakpoint(0xF02C);
5.      ...
6.   }
```

2. __nop：此内在函数在编译器生成的指令流中插入 NOP 指令或等效的代码序列。将为源代码中的每个__nop 内在函数生成一个 NOP 指令。

注：编译器不会优化删除 NOP 指令。

语法：

[cpp] view plaincopy

```
1.  void __nop(void)
```