The Linux Macro Journey

Version 1.0
August-2024

By Dr. Shlomi Boutnaru

Created using Craiyon, AI Image Generator

Table of Contents

Introduction

When starting to read the source code of the Linux kernel I believe that they are basic macro that everyone needs to know about. Because of that I have decided to write a series of short writeups aimed at providing the basic vocabulary and understanding for achieving that.

Overall, I wanted to create something that will improve the overall knowledge of Linux kernel in writeups that can be read in 1-3 mins. I hope you are going to enjoy the ride.

Lastly, you can follow me on twitter - @boutnaru (https://twitter.com/boutnaru). Also, you can read my other writeups on medium - https://medium.com/@boutnaru. Lastly, You can find my free eBooks at https://TheLearningJourneyEbooks.com.

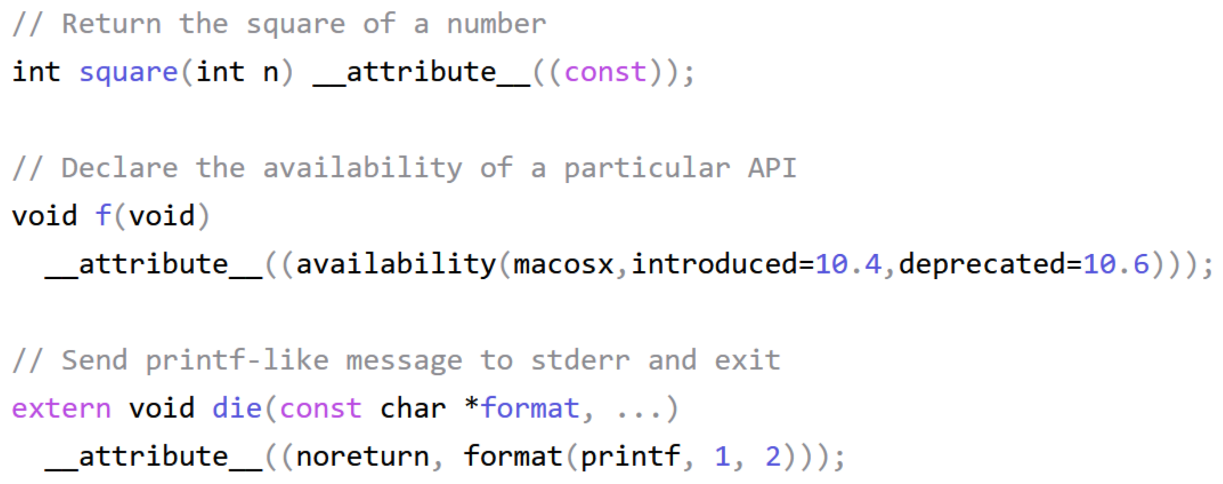
Lets GO!!!!!!

__attribute__

When using gcc we can use the keyword "__attribute__" which can specify special attributes for variables[1] or even functions[2]. The "__attribute__" keyword is followed by an attribute specification inside double parentheses like: "__attribute__(([SOME_ATTRIBUTE]))" - as shown in the example below[3].

Overall, they are different attributes that can be specified. A couple of examples are: "section" (which states the location in the binary the function/variable should be placed), "weak" (which defines a weak symbol) and "aligned" (which states that the variable must be aligned to a specific number of bytes). It is important to know that clang also supports "__attribute__"[4]. By the way, there is different support for attributes between gcc and clang.

Lastly, we can say that the "__attribute__" mechanism allows developers to attach characteristics to functions/variables that enables the compiler to perform extra error checking/optimizations/controlling the layout of the compiler binary and more[5]. Also, in the source code of the Linux kernel (version 6.5) there are 1027 files in which "__attribute__" is used[6]. It was first used in kernel 2.5.21[7].



```
// Return the square of a number
int square(int n) __attribute__((const));

// Declare the availability of a particular API
void f(void)
  __attribute__((availability(macosx,introduced=10.4,deprecated=10.6)));

// Send printf-like message to stderr and exit
extern void die(const char *format, ...)
  __attribute__((noreturn, format(printf, 1, 2)));
```

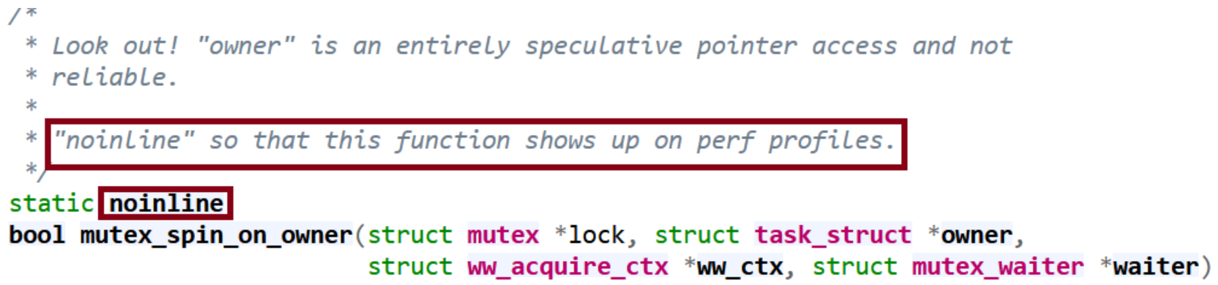
[1] https://www.linuxtopia.org/online_books/programming_tool_guides/linux_using_gnu_compiler_collection/variable-attributes.html
[2] https://www.linuxtopia.org/online_books/programming_tool_guides/linux_using_gnu_compiler_collection/function-attributes.html
[3] https://nshipster.com/__attribute__/
[4] https://clang.llvm.org/docs/AttributeReference.html
[5] http://www.unixwiz.net/techtips/gnu-c-attributes.html
[6] https://elixir.bootlin.com/linux/v6.5/C/ident/__attribute__
[7] https://elixir.bootlin.com/linux/v2.5.21/C/ident/__attribute__

noinline

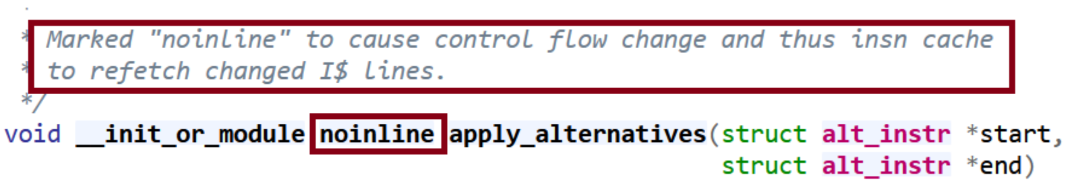
The “noinline” macro is based on “__attribute__”[8], it is defined as part of the “compiler_attributes.h”[9]. By using “noinline” it prevents a function from being considered for inlining. One reason for using “noinline” is to keep functions called from the _init section from being discarded while they're still in use[10].

Thus, a non-inline function is not expanded in place at the call site. The compiler will call the function as usual and won’t copy the body of the function into the code at the point where it is called[11]. As of kernel 6.7 there are 299 references in code files using the “noinline” macro[12]. This is done in different areas of the Linux kernel source code such as: drivers, file systems, memory management, network stack, security and more.

Lastly, “noinline” is supported both by gcc and by clang. We can use it in order to suppresses the inlining of a function at the call sites of the function[13]. There are other use cases that “noinline” can be used like: changing flow control and ensuring a function shows up on perf profiles - as shown in the screenshot below taken from the Linux source code.



```
/*
 * Look out! "owner" is an entirely speculative pointer access and not
 * reliable.
 *
 * "noinline" so that this function shows up on perf profiles.
 */
static noinline
bool mutex_spin_on_owner(struct mutex *lock, struct task_struct *owner,
                         struct ww_acquire_ctx *ww_ctx, struct mutex_waiter *waiter)
```



```
 * Marked "noinline" to cause control flow change and thus insn cache
 * to refetch changed I$ lines.
 */
void __init_or_module noinline apply_alternatives(struct alt_instr *start,
                                                  struct alt_instr *end)
```

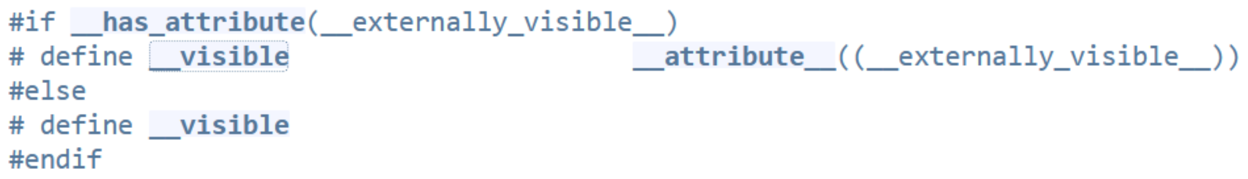
[8] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[9] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L244
[10] https://www.kernel.org/doc/local/inline.html
[11] https://aviatesk.github.io/posts/inlining-101/#so_what_is_inlining
[12] https://elixir.bootlin.com/linux/v6.7/C/ident/noinline
[13] https://clang.llvm.org/docs/AttributeReference.html#noinline

__visible

The "__visible" macro is based on "__attribute__"[14], it is defined as part of the "compiler_attributes.h" - as shown in the code snippet below[15]. The goal of "__visible" is to allow an object (variable/function) to remain visible outside the current compilation unit[16].

Overall, we can say that "__visible" nullifies gcc's command line option "-fwhole-program". This option is part of the optimization mechanism of gcc, it assumes the current compilation unit represents the whole program being compiled. Thus, all public variable/functions (excluding "main" and those marked "__visible") become static functions and lead to a more aggressive interprocedural optimization[17].

Lastly, "__visible" is supported only by gcc and not by clang, thus we can look at it as optional[18]. We can check out all the references of "__visible" as for kernel version 6.7 for more insights[19].



```
#if __has_attribute(__externally_visible__)
# define __visible                      __attribute__((__externally_visible__))
#else
# define __visible
#endif
```

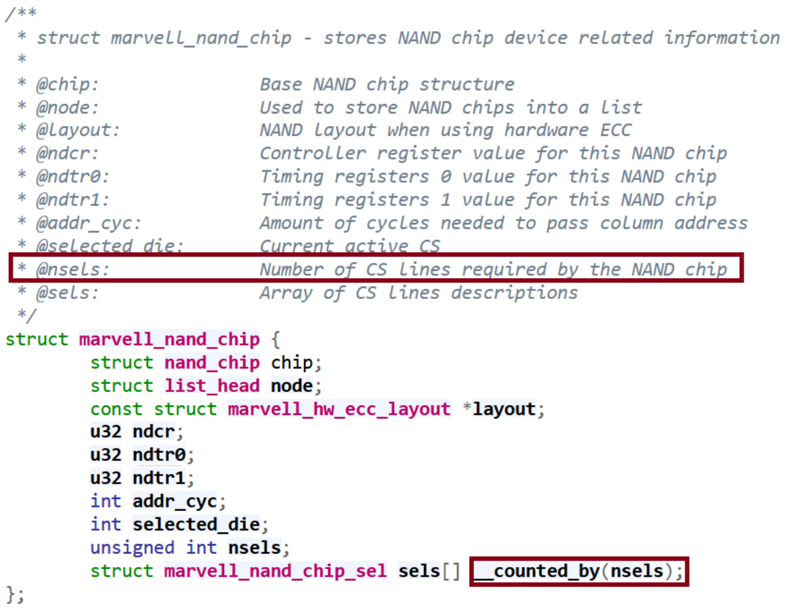
[14] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[15] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L162
[16] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-externally_005fvisible-function-attribute
[17] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[18] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L157
[19] https://elixir.bootlin.com/linux/v6.7/C/ident/__visible

__counted_by

The "__counted_by" macro is based on "__attribute__"[20], it is defined as part of the "compiler_attributes.h"[21]. "__counted_by" is used on flexible array members, it gets as a argument the field name (in the holding structure) which holds the count of the elements in the flexible array[22].

Overall, the information provided by "__counted_by" is used for array bounds sanitizers (for security reasons). This attribute is optional and relevant when using gcc >= 14[23] or clang >=18[24].

Lastly, "__counted_by" is used by different data structures related like hardware drivers (NICs, SOCs, USB, GPU, sound, storage devices and more) and security components across the Linux kernel code[25]. One example is Marvell's NAND chip controller driver - as shown in the code snippet below[26].



```
/**
 * struct marvell_nand_chip - stores NAND chip device related information
 *
 * @chip:               Base NAND chip structure
 * @node:               Used to store NAND chips into a list
 * @layout:             NAND layout when using hardware ECC
 * @ndcr:               Controller register value for this NAND chip
 * @ndtr0:              Timing registers 0 value for this NAND chip
 * @ndtr1:              Timing registers 1 value for this NAND chip
 * @addr_cyc:           Amount of cycles needed to pass column address
 * @selected_die:       Current active CS
 * @nsels:              Number of CS lines required by the NAND chip
 * @sels:               Array of CS lines descriptions
 */
struct marvell_nand_chip {
        struct nand_chip chip;
        struct list_head node;
        const struct marvell_hw_ecc_layout *layout;
        u32 ndcr;
        u32 ndtr0;
        u32 ndtr1;
        int addr_cyc;
        int selected_die;
        unsigned int nsels;
        struct marvell_nand_chip_sel sels[] __counted_by(nsels);
};
```

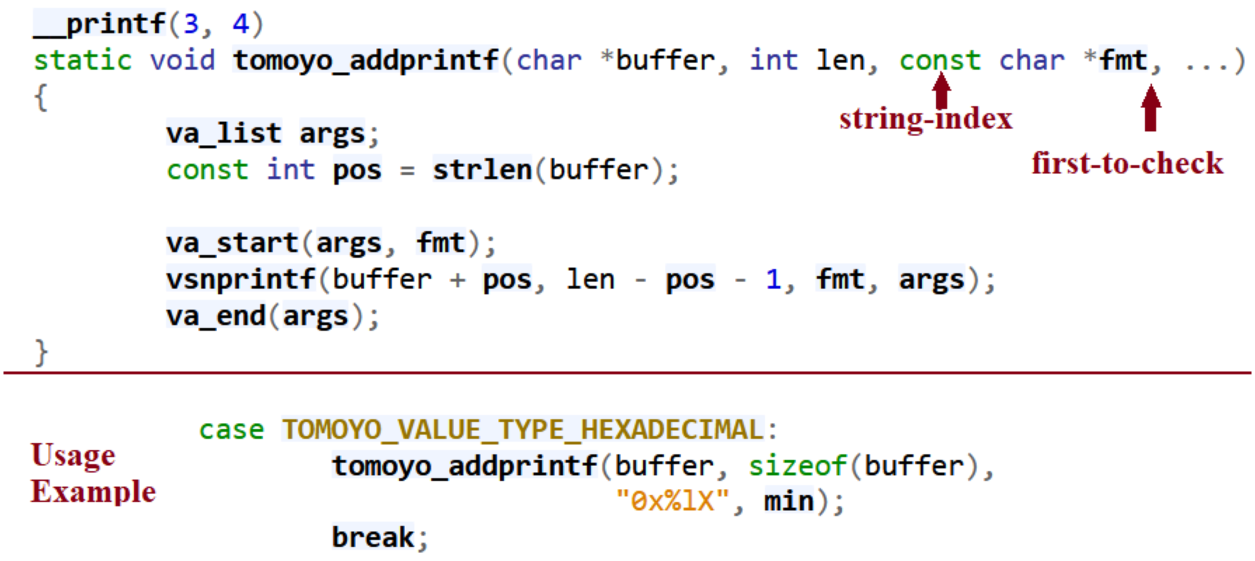
[20] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[21] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L104
[22] https://web.archive.org/web/20230928010656/https://reviews.llvm.org/D148381
[23] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L98
[24] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L99
[25] https://elixir.bootlin.com/linux/v6.7/A/ident/__counted_by
[26] https://elixir.bootlin.com/linux/v6.7/source/drivers/mtd/nand/raw/marvell_nand.c#L351

__printf

The “__printf” macro is based on “__attribute__”[27], it is defined as part of the “compiler_attributes.h”[28]. As for kernel version 6.8.1, there are 289 files which reference the usage of “__printf”[29].

Overall, “__printf” is used to specify the function that takes “printf” style arguments. Those arguments should be type-checked against a format string. By using it the compiler checks the consistency of the argument with the format string style[30]. This feature is supported both by gcc and clang compilers[31].

Lastly, the attribute takes two parameters the first is “string-index” which is the index for the format string like string and “first-to-check” which is the first argument we want to start the type-checking - example form the kernel source code is shown below both the definition of the function[32] and a usage example[33].



```
__printf(3, 4)
static void tomoyo_addprintf(char *buffer, int len, const char *fmt, ...)
{
        va_list args;
        const int pos = strlen(buffer);

        va_start(args, fmt);
        vsnprintf(buffer + pos, len - pos - 1, fmt, args);
        va_end(args);
}
```

```
    case TOMOYO_VALUE_TYPE_HEXADECIMAL:
            tomoyo_addprintf(buffer, sizeof(buffer),
                             "0x%lX", min);
            break;
```

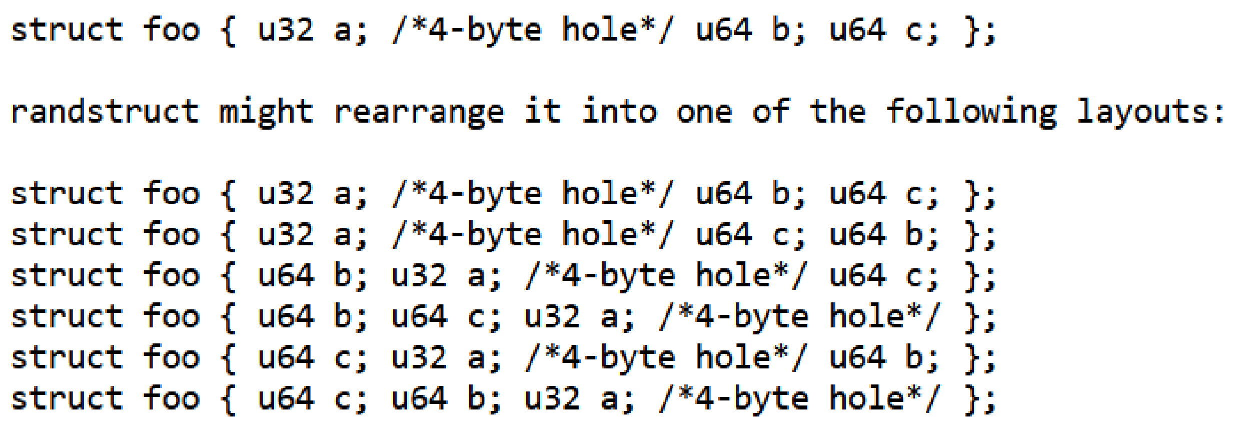
[27] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[28] https://elixir.bootlin.com/linux/v6.8.1/source/include/linux/compiler_attributes.h#L171
[29] https://elixir.bootlin.com/linux/v6.8.1/C/ident/__printf
[30] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-format-function-attribute
[31] https://clang.llvm.org/docs/AttributeReference.html#format
[32] https://elixir.bootlin.com/linux/v6.8.1/source/security/tomoyo/common.c#L187
[33] https://elixir.bootlin.com/linux/v6.8.1/source/security/tomoyo/common.c#L437

__randomize_layout

Overall, "__randomize_layout" is macro which defined in the Linux source code as part of the "compiler_type.h" file[34]. It is based on the RANDSTRUCT gcc plugin[35].

Moreover, RANDSTRUCT is a gcc compiler that was ported from grsecurity to the upstream kernel[36]. Its goal is to provide structure randomization in the kernel - as shown in the example below[37]. Since kernel 4.8, gcc's plugin infrastructure has been used by the Linux kernel in order to implement such support for KSPP (Kernel Self Protection Project). KSPP ported features from grsecurity/PaX for hardaning the mainline kernel[38].

Also, it is known as the randomized layout of sensitive kernel structures which is controlled using the configuration item "CONFIG_GCC_PLUGIN_RANDSTRUCT". If enabled the layout of the structures that are entirely function pointers (and are not marked as "__no_randomize_layout"), or structures that are marked as "__randomize_layout" are going to be randomized at compiled time[39]. Lasly, there are different data structures that are explicitly marked with "__randomize_layout" like: "struct cred"[40], "struct vm_area_struct"[41] and "struct vsmount"[42].



```
struct foo { u32 a; /*4-byte hole*/ u64 b; u64 c; };

randstruct might rearrange it into one of the following layouts:

struct foo { u32 a; /*4-byte hole*/ u64 b; u64 c; };
struct foo { u32 a; /*4-byte hole*/ u64 c; u64 b; };
struct foo { u64 b; u32 a; /*4-byte hole*/ u64 c; };
struct foo { u64 b; u64 c; u32 a; /*4-byte hole*/ };
struct foo { u64 c; u32 a; /*4-byte hole*/ u64 b; };
struct foo { u64 c; u64 b; u32 a; /*4-byte hole*/ };
```

[34] https://elixir.bootlin.com/linux/v6.4.11/source/include/linux/compiler_types.h#L293
[35] https://github.com/torvalds/linux/blob/master/scripts/gcc-plugins/randomize_layout_plugin.c
[36] https://github.com/clang-randstruct/plugin
[37] https://www.spinics.net/lists/kernel-hardening/msg05669.html
[38] https://lwn.net/Articles/722293/
[39] https://cateee.net/lkddb/web-lkddb/GCC_PLUGIN_RANDSTRUCT.html
[40] https://elixir.bootlin.com/linux/v6.4.11/source/include/linux/cred.h#L153
[41] https://elixir.bootlin.com/linux/v6.4.11/source/include/linux/mm_types.h#L588
[42] https://elixir.bootlin.com/linux/v6.4.11/source/include/linux/mount.h#L75

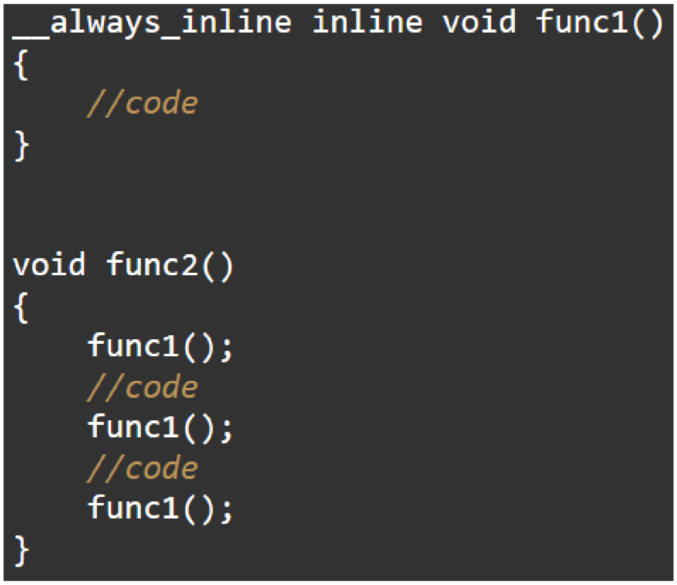
__always_inline

The “__always_inline” macro is based on “__attribute__”[43], it is defined as part of the “compiler_attributes.h”[44]. As of kernel 6.7 the “__always_inline” macro is referenced in 677 different source files[45].

Overall, functions usually aren't inlined unless optimization is turned on. “__always_inline” forces inlining even if standard restrictions apply (it's an error if inlining fails). Indirect calls to those functions might be inlined or not, depending on compiler settings and optimization level[46].

Moreover, even though folks are using “__always_inline”, they're not explicitly writing "inline" with it. However, it is necessary for GCC to apply the attribute as intended[47]. Without using “inline” gcc will display a warning message (unless disabled - [-Wattributes]): “warning: 'always_inline' function might not be inlinable”[48].

Lastly, there are use cases in which “__always_inline” has more cons than pros (which removes function call overhead but has some limitations). Think about a function that calls multiple times to a function marked as “__always_inline” - as shown in the pseudo code below. In this case we get the cons of inlining: larger program size, longer build time and worse cache locality[49].



```
__always_inline inline void func1()
{
    //code
}


void func2()
{
    func1();
    //code
    func1();
    //code
    func1();
}
```

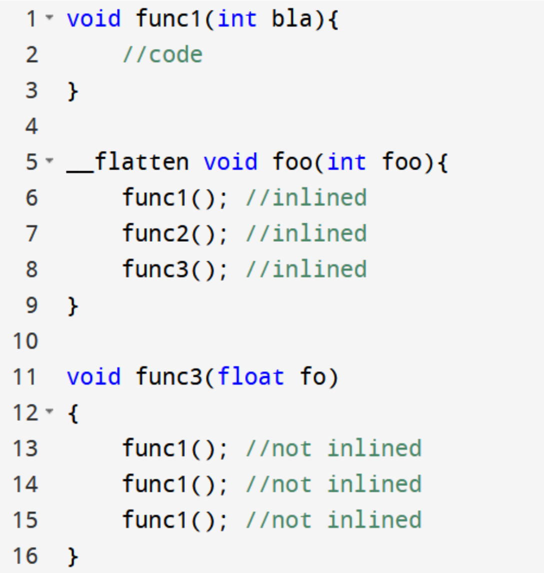
[43] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[44] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L55
[45] https://elixir.bootlin.com/linux/v6.7/A/ident/__always_inline
[46] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-always_005finline-function-attribute
[47] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L47
[48] https://stackoverflow.com/questions/32432596/warning-always-inline-function-might-not-be-inlinable-wattributes
[49] https://awesomekling.github.io/Smarter-C++-inlining-with-attribute-flatten/

__flatten

The "__flatten" macro is based on "__attribute__"[50], it is defined as part of the "compiler_attributes.h"[51]. As of kernel 6.7 the "__flatten" macro is referenced in 9 different source files[52].

Moreover, in case a function is marked as "__flatten", every call inside that function is inlined. This includes the calls that such inlining introduces to the function. However, this excludes recursive calls to the function itself[53]. Thus, all the callees of the function are inlined into it[54]. This allows us to overcome some of the limitations caused by using "__always_inline"[55] - as shown in the pseudo code below.

Lastly, there are cases which cannot be inlined (even if they are marked as "__flatten"). Examples are if the body of the callee is unavailable or the callee has an "noninline"[56] attribute[57].



```
void func1(int bla){
    //code
}

__flatten void foo(int foo){
    func1(); //inlined
    func2(); //inlined
    func3(); //inlined
}

void func3(float fo)
{
    func1(); //not inlined
    func1(); //not inlined
    func1(); //not inlined
}
```

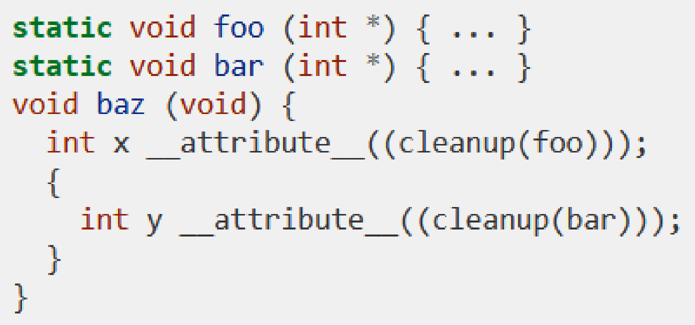
[50] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[51] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L236
[52] https://elixir.bootlin.com/linux/v6.7/A/ident/__flatten
[53] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes
[54] https://awesomekling.github.io/Smarter-C++-inlining-with-attribute-flatten/
[55] https://medium.com/@boutnaru/the-linux-kernel-macro-journey-always-inline-d707e230b9ee
[56] https://medium.com/@boutnaru/the-linux-kernel-macro-jour-8232e8f9b3bb
[57] https://clang.llvm.org/docs/AttributeReference.html#flatten

__cleanup

The "__cleanup" macro is based on "__attribute__"[58], it is defined as part of the "compiler_attributes.h" in the Linux source code[59].

Overall, the "__cleanup" macro is used for running a specific function when a local variable goes out of scope[60] -as shown in the example below. This feature is supported both in clang and in gcc[61].

Lastly, the variable which goes out of scope is given as a parameter to the call back function defined using "__cleanup". Also, in case we have two variables in the same scope their call back functions are called in reverse order to their declarations.



```
static void foo (int *) { ... }
static void bar (int *) { ... }
void baz (void) {
  int x __attribute__((cleanup(foo)));
  {
    int y __attribute__((cleanup(bar)));
  }
}
```

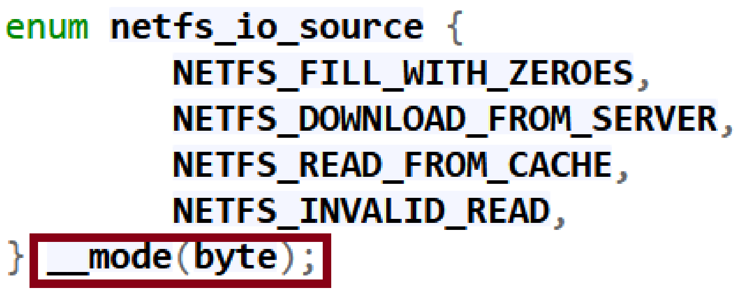
[58] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[59] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L76
[60] https://clang.llvm.org/docs/AttributeReference.html#cleanup
[61] https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-cleanup-variable-attribute

__mode

The "__mode" macro is based on "__attribute__"[62], it is defined as part of the "compiler_attributes.h"[63]. "__mode" is used for specifying the data type for the declaration (which is passed to the macro). Thus it lets us request an integer or floating-point type according to its width[64].

Moreover, in order to checkout the list of possible keywords for "__mode" we can go over the "Machine Modes" as part of the GCC internals[65]. We can checkout an example of usage taken from the Linux kernel source code in the screenshot below[66]. The term "byte" used below refers to an object of BITS_PER_UNIT bits[67].

Lastly, as of kernel version 6.7 "__mode" is defined as part of a variable in 31 files and in 4 files it is part of a typedef[68]. By the way, clang also supports the mode compiler attribute[69].



```
enum netfs_io_source {
        NETFS_FILL_WITH_ZEROES,
        NETFS_DOWNLOAD_FROM_SERVER,
        NETFS_READ_FROM_CACHE,
        NETFS_INVALID_READ,
} __mode(byte);
```

[62] https://medium.com/@boutnaru/the-attribute-keyword-2ee44f59ab25
[63] https://elixir.bootlin.com/linux/v6.7/source/include/linux/compiler_attributes.h#L190
[64] https://gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-mode-type-attribute
[65] https://gcc.gnu.org/onlinedocs/gccint/Machine-Modes.html#Machine-Modes
[66] https://elixir.bootlin.com/linux/v6.7/source/include/linux/netfs.h#L112
[67] https://gcc.gnu.org/onlinedocs/gccint/Storage-Layout.html
[68] https://elixir.bootlin.com/linux/v6.7/C/ident/__mode
[69] https://releases.llvm.org/15.0.0/tools/clang/docs/AttributeReference.html