1. printk infrastructure

Some subsystems have their own custom printk that applies a va_format to a generic format, for example, to include a device number or other metadata alongside the format supplied by the caller.

In order to store these in the way they would be emitted by the printk infrastructure, the subsystem provides us with the start, fixed string, and any subsequent text in the format string.

We take a variable argument list as pr_fmt/dev_fmt/etc are sometimes passed as multiple arguments (eg: "%s: ", "blah"), and we must only take the first one.

subsys_fmt_prefix must be known at compile time, or compilation will fail (since this is a mistake). If fmt or level is not known at compile time, no index entry will be made (since this can legitimately happen).

```
#define __printk_index_emit( . . . ) do { } while (0)
#define printk_index_wrap(_p_func, _fmt, . . . ) (
        {
          __printk_index_emit(_fmt, Λ, Λ);
          _p_func(_fmt, ##__VA_ARGS__);
        }
        )
```

2. printk()

printk() - print a kernel message.

fmt: format string.

. . . : arguments for the format string.

This is printk(). It can be called from any context. We want it to work.

If printk indexing is enabled, _printk() is called from printk_index_wrap. Otherwise, printk is simply

#defined to _printk.

We try to grab the console_lock. If we succeed, it's easy - we log the output and call the console drivers. If we fail to get the semaphore, we place the output into the log buffer and return. The current holder of the console_sem will notice the new output in console_unlock(); and will send it to the consoles before releasing the lock.

One effect of this deferred printing is that code which calls printk() and then changes console_loglevel may break. This is because console_loglevel is inspected when the actual printing occurs.

See also:

printf(3)

See the vsnprintf() documentation for format string extensions over C99.

```
#define printk(fmt, . . . ) printk_index_wrap(_printk, fmt, ##__VA_ARGS__)
#define pr_fmt(fmt) "VUnit:␣"fmt
#define pr_info(fmt, . . . ) printk(pr_fmt(fmt), ##__VA_ARGS__)
```

3. WARN(), WARN_ON() and so on can be used to report significant kernel issues that need prompt attention if they should ever appear at runtime.

Do not use these macros when checking for invalid external inputs.

```
#define __WARN_printf(arg ...) do
        {
          fprintf(stderr, arg);
        }
        while (0)
#define WARN(condition, format ...) (
        {
          int __ret_warn_on = ¬¬(condition);
          if (unlikely(__ret_warn_on))
            printf(format);
          unlikely(__ret_warn_on);
        }
        )
#define WARN_ONCE(condition, format ...) WARN(condition, format)
#define WARN_ON(condition) (
        {
          int __ret_warn_on = ¬¬(condition);
          if (unlikely(__ret_warn_on))
            __WARN_printf("assertion␣failed␣at␣%s:%d\n",
            __FILE__, __LINE__);
          unlikely(__ret_warn_on);
        }
        )
```

4. Don't use BUG() or BUG_ON() unless there's really no way out; one example might be detecting data structure corruption in the middle of an operation that can't be backed out of. If the (sub)system can somehow continue operating, perhaps with reduced functionality, it's probably not BUG-worthy.

If you're tempted to BUG(), think again: is completely giving up really the **only** solution? There are usually better options, where users don't need to reboot ASAP and can mostly shut down cleanly.

```
#define BUG_ON(cond) OS_TRAP_IF(¬(cond))
#define BUG() BUG_ON(1)
```

5. Comparing types.

Are two types/vars the same type (ignoring qualifiers)?

```
#define __same_type(a, b) __builtin_types_compatible_p(typeof(a), typeof(b))
```

6. Multiplication with overflow.

```
#define check_mul_overflow(a, b, d) __must_check_overflow((
        {
          typeof(a)__a = (a);
          typeof(b)__b = (b);
          typeof(d)__d = (d);
          (void)(&__a ≡ &__b);
          (void)(&__a ≡ __d);
          __builtin_mul_overflow(__a, __b, __d);
        }
        ))
```

7. The **volatile** is due to gcc bugs: see Extended Asm

With extended asm you can read and write C variables from assembler and perform jumps from assembler code to C labels. Extended asm syntax uses colons (:) to delimit the operand parameters after the assembler template. See Assembly with C

In it find recommondations: Barrier Sync

The first barriers does nothing at runtime. It's called a SW barrier. The second barrier would translate into a HW barrier, probably a fence (mfence/sfence) operations if you're on x86. This instruction tells the micro code of a CPU network to make sure that loads or stores can't pass this point and must be observed in the correct side of the sync point.

```
#define barrier() __asm____volatile__ ( "": : : "memory" )
```

8. Prevent the compiler from merging or refetching reads or writes. The compiler is also forbidden from reordering successive instances of READ_ONCE and WRITE_ONCE, but only when the compiler is aware of some particular ordering. One way to make the compiler aware of ordering is to put the two invocations of READ_ONCE or WRITE_ONCE in different C statements.

These two macros will also work on aggregate data types like structs or unions. If the size of the accessed data type exceeds the word size of the machine (e.g., 32 bits or 64 bits) READ_ONCE() and WRITE_ONCE() will fall back to memcpy and print a compile-time warning.

Their two major use cases are: (1) Mediating communication between process-level code and irq/NMI handlers, all running on the same CPU, and (2) Ensuring that the compiler does not fold, spindle, or otherwise mutilate accesses that either do not require ordering or that interact with an explicit memory barrier or atomic instruction that provides the required ordering.

```
#define READ_ONCE(x) ( {
        union {
          typeof (x)__val;
          char __c[1];
        } __u =
        { . __c = {0} } ;
        __read_once_size(&(x), __u.__c, sizeof (x));
        __u.__val;
        } )
#define WRITE_ONCE(x, val) ( {
        union {
          typeof (x)__val;
          char __c[1];
        } __u =
        { . __val = (val) } ;
        __write_once_size(&(x), __u.__c, sizeof (x));
        __u.__val;
        } )
```

9. Print a refcount warning like "underflow; use-after-free".

```
#define REFCOUNT_WARN(str) WARN_ONCE(1, "refcount_t:␣"str".\n")
```

10. Built-in Function: *long __builtin_expect (long exp, long c)*

See Builtin_Expect

You may use __builtin_expect to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (-fprofile-arcs), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

3The return value is the value of exp, which should be an integral expression. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if (__builtin_expect(x, 0))
    foo()
```

indicates that we do not expect to call foo, since we expect x to be zero. Since you are limited to integral expressions for exp, you should use constructions such as

```
if (__builtin_expect(ptr! = NULL, 1))
    foo(*ptr)
```

when testing pointer or floating-point values.

For the purposes of branch prediction optimizations, the probability that a –builtinexpect expression is true is controlled by GCCs builtin-expect-probability parameter, which defaults to 90

You can also use –builtin-expect-with-probability to explicitly assign a probability value to individual expressions. If the built-in is used in a loop construct, the provided probability will influence the expected number of iterations made by loop optimizations.

11. The attributes likely and unlikely may be applied to labels and statements (other than declaration-statements). They may not be simultaneously applied to the same label or statement.

1 Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are more likely than any alternative path of execution that does not include such a statement.

2 Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are less likely than any alternative path of execution that does not include such a statement.

```
#define unlikely(x) __builtin_expect(¬¬(x), 0)
```

12. static_assert - check integer constant expression at build time

static_assert() is a wrapper for the C11 _Static_assert, with a little macro magic to make the message optional (defaulting to the stringification of the tested expression).

Contrary to *BUILD_BUG_ON()*, *static_assert()* can be used at global scope, but requires the expression to be an integer constant expression (i.e., it is not enough that *__builtin_constant_p()* is true for expr).

Also note that *BUILD_BUG_ON()* fails the build if the condition is true, while *static_assert()* fails the build if the expression is false.

```
#define static_assert(expr, ...) __static_assert(expr, ##__VA_ARGS__, #expr)
#define __static_assert(expr, msg, ...) _Static_assert(expr, msg)
```

13. container_of - cast a member of a structure out to the containing structure.

ptr:	the pointer to the member.
type:	the type of the container struct this is embedded in.
member:	the name of the member within the struct.

```
#define container_of(ptr, type, member) ( {
        void *__mptr = (void *)(ptr);
        static_assert ( __same_type (*(ptr), ( ( type * ) 0 ) → member ) ∨
        __same_type(*(ptr), void),
        "pointer␣type␣mismatch␣in␣container_of()" ) ;
        ( ( type * ) (__mptr − offsetof(type, member)) ) ; } )
```

14. list_entry - get the struct for this entry.

ptr: the *&struct list_head* pointer.
type: the type of the struct this is embedded in.
member: the name of the list_head within the struct.

#define *list_entry*(*ptr*, *type*, *member*)
container_of(*ptr*, *type*, *member*)

15. list_prev_entry - get the prev element in list.

pos: the type * to cursor.
member: the name of the list_head within the struct.

#define *list_prev_entry*(*pos*, *member*)
list_entry((*pos*)→*member*.*prev*, *typeof*(∗(*pos*)), *member*)

16. list_next_entry - get the next element in list.

pos: the type * to cursor.
member: the name of the list_head within the struct.

#define *list_next_entry*(*pos*, *member*)
list_entry((*pos*)→*member*.*next*, *typeof*(∗(*pos*)), *member*)

17. list_last_entry - get the last element from a list.

ptr: the list head to take the element from.
type: the type of the struct this is embedded in.
member: the name of the list_head within the struct.

#define *list_last_entry*(*ptr*, *type*, *member*)
list_entry((*ptr*)→*prev*, *type*, *member*)

18. list_first_entry - get the first element from a list.

ptr: the list head to take the element from.
type: the type of the struct this is embedded in.
member: the name of the list_head within the struct.

#define *list_first_entry*(*ptr*, *type*, *member*)
list_entry((*ptr*)→*next*, *type*, *member*)

19. list_entry_is _head - test if the entry points to the head of the list.

pos: the type * to cursor.
head: the head for your list.
member: the name of the list_head within the struct.

#define *list_entry_is_head*(*pos*, *head*, *member*)
(&*pos*→*member* ≡ (*head*))

20. list_for_each_entry - iterate over list of given type.

pos: the type * to cursor.
head: the head for your list.
member: the name of the list_head within the struct.

#define *list_for_each_entry*(*pos*, *head*, *member*)
for (*pos* = *list_first_entry*(*head*, *typeof*(∗*pos*), *member*);
¬*list_entry_is_head*(*pos*, *head*, *member*);
pos = *list_next_entry*(*pos*, *member*))

21. list_for_each_entry_reverse - iterate backwards over list of given type.

pos:	the type * to cursor.
head:	the head for your list.
member:	the name of the list_head within the struct.

```
#define list_for_each_entry_reverse(pos, head, member)
        for (pos = list_last_entry(head, typeof(∗pos), member);
        ¬list_entry_is_head(pos, head, member);
        pos = list_prev_entry(pos, member))
```

22. list_for_each_entry_safe - iterate over list of given type safe against removal of list entry.

pos:	the type * to use as a loop cursor.
n:	another type * to use as temporary storage.
head:	the head for your list.
member:	the name of the list_head within the struct.

```
#define list_for_each_entry_safe(pos, n, head, member)
        for (pos = list_first_entry(head, typeof(∗pos), member),
        n = list_next_entry(pos, member);
        ¬list_entry_is_head(pos, head, member);
        pos = n, n = list_next_entry(n, member))
```

23. Index.

MACROS