printk()在kernel initialization中很早期就可以调用。只要进入start_kernel()以后就可以调用printk()了。在start_kernel()以前的汇编代码运行时，由于还没有初始化好kernel的c的运行环境，所以是不能调用printk()。

ARM kernel中，在start_kernel()以前，也不能调用arch/arm/kernel/setup.c中的early_print()，因为early_print()也依赖于下面两个约束条件：

1. c的运行环境

2. 输出log的串口的virtual address的mapping是在调用machine descriptor的.map_io() callback function以后才建立的，所以也不能调用。

在Gr2 / Gs2 LSP中

in arch/arm/mach-pegmatite/pegmatite.c

```
DT_MACHINE_START(PEGMATITE_DT, "Marvell Pegmatite (Device Tree)")
#ifdef CONFIG_SMP
        .smp            = smp_ops(pegmatite_smp_ops),
#endif
        .init_machine  = pegmatite_dt_init,
        .map_io         = pegmatite_map_io,
        .init_early = pegmatite_init_early,
        .init_irq    = pegmatite_init_irq,
        .init_time  = pegmatite_timer_and_clk_init,
        .restart    = pegmatite_restart,
        .dt_compat     = pegmatite_dt_compat,
#ifdef CONFIG_ZONE_DMA
        .dma_zone_size     = SZ_256M,
#endif
MACHINE_END

static struct map_desc pegmatite_io_desc[] __initdata = {
```

```
    {
        .virtual    = (unsigned long) PEGMATITE_REGS_VIRT_BASE,
        .pfn        = __phys_to_pfn(PEGMATITE_REGS_PHYS_BASE),
        .length         = PEGMATITE_REGS_SIZE,
        .type       = MT_DEVICE,
    },
    {
        .virtual    = (unsigned long) PEGMATITE_UPC_VIRT_BASE,
        .pfn        = __phys_to_pfn(PEGMATITE_UPC_PHYS_BASE),
        .length     = 0x000C0000,
        .type           = MT_DEVICE
    },
};

void __init pegmatite_map_io(void)
{
    iotable_init(pegmatite_io_desc, ARRAY_SIZE(pegmatite_io_desc));
}
```

所以在pegmatite_map_io（）被调用以前要输出debug log比较费劲，可能只能用JTAG来汇编级调试了。

==========

printk()虽然可以在进入start_kernel（）以后就可以被调用，但真正可以输出log那是在start_kernel()中调用console_init()以后。期间的所有log实际上都被缓存在log buffer中，并不会从串口输出。所以要调试console_init()以前的code，要调用early_print()。该函数直接写串口，developer可以马上看到log。

printk()

|

|

\|/

vprintk()

|

|

\|/

printk_emit()

|

|

\|/

vprintk_emit()

in vprintk_emit() function

```
        /*
         * Try to acquire and then immediately release the console
         * semaphore.  The release will print out buffers and wake up
         * /dev/kmsg and syslog() users.
         */
        if (console_trylock_for_printk())
console_unlock();
```

```
/**
 * console_unlock - unlock the console system
 *
 * Releases the console_lock which the caller holds on the console system
 * and the console driver list.
 *
 * While the console_lock was held, console output may have been buffered
 * by printk().  If this is the case, console_unlock(); emits
 * the output prior to releasing the lock.
 *
 * If there is output waiting, we wake /dev/kmsg and syslog() users.
 *
 * console_unlock(); may be called from any context.
 */
void console_unlock(void)
{
        static char text[LOG_LINE_MAX + PREFIX_MAX];
        static u64 seen_seq;
        unsigned long flags;
        bool wake_klogd = false;
        bool retry;

        if (console_suspended) {
                up_console_sem();
                return;
        }

        console_may_schedule = 0;

        /* flush buffered message fragment immediately to console */
        console_cont_flush(text, sizeof(text));
again:
        for (;;) {
                struct printk_log *msg;
                size_t len;
```

```
36.                     int level;
37.
38.                     raw_spin_lock_irqsave(&logbuf_lock, flags);
39.                     if (seen_seq != log_next_seq) {
40.                             wake_klogd = true;
41.                             seen_seq = log_next_seq;
42.                     }
43.
44.                     if (console_seq < log_first_seq) {
45.                             len = sprintf(text, "** %u printk messages dropped ** ",
46.                                           (unsigned)(log_first_seq - console_seq));
47.
48.                             /* messages are gone, move to first one */
49.                             console_seq = log_first_seq;
50.                             console_idx = log_first_idx;
51.                             console_prev = 0;
52.                     } else {
53.                             len = 0;
54.                     }
55.     skip:
56.                     if (console_seq == log_next_seq)
57.                             break;
58.
59.                     msg = log_from_idx(console_idx);
60.                     if (msg->flags & LOG_NOCONS) {
61.                             /*
62.                              * Skip record we have buffered and already printed
63.                              * directly to the console when we received it.
64.                              */
65.                             console_idx = log_next(console_idx);
66.                             console_seq++;
67.                             /*
68.                              * We will get here again when we register a new
69.                              * CON_PRINTBUFFER console. Clear the flag so we
70.                              * will properly dump everything later.
71.                              */
```

```
72.                             msg->flags &= ~LOG_NOCONS;
73.                             console_prev = msg->flags;
74.                             goto skip;
75.                     }
76.
77.                     level = msg->level;
78.                     len += msg_print_text(msg, console_prev, false,
79.                                           text + len, sizeof(text) - len);
80.                     console_idx = log_next(console_idx);
81.                     console_seq++;
82.                     console_prev = msg->flags;
83.                     raw_spin_unlock(&logbuf_lock);
84.
85.                     stop_critical_timings();        /* don't trace print latency */
86.                     call_console_drivers(level, text, len);
87.                     start_critical_timings();
88.                     local_irq_restore(flags);
89.             }
90.             console_locked = 0;
91.
92.             /* Release the exclusive_console once it is used */
93.             if (unlikely(exclusive_console))
94.                     exclusive_console = NULL;
95.
96.             raw_spin_unlock(&logbuf_lock);
97.
98.             up_console_sem();
99.
100.            /*
101.             * Someone could have filled up the buffer again, so re-check if there's
102.             * something to flush. In case we cannot trylock the console_sem again,
103.             * there's a new owner and the console_unlock() from them will do the
104.             * flush, no worries.
105.             */
106.            raw_spin_lock(&logbuf_lock);
107.            retry = console_seq != log_next_seq;
```

```
108.            raw_spin_unlock_irqrestore(&logbuf_lock, flags);
109.
110.            if (retry && console_trylock())
111.                    goto again;
112.
113.            if (wake_klogd)
114.                    wake_up_klogd();
115.    }
116.    EXPORT_SYMBOL(console_unlock);
```

如果console不能获得，则printk()的输出只能是buffer在log buffer中。

真正的输出log在call_console_drivers()中

```
/*
 * Call the console drivers, asking them to write out
 * log_buf[start] to log_buf[end - 1].
 * The console_lock must be held.
 */
static void call_console_drivers(int level, const char *text, size_t len)
{
        struct console *con;

        trace_console(text, len);

        if (level >= console_loglevel && !ignore_loglevel)      ①
                return;
        if (!console_drivers)                                                ②
return;

        for_each_console(con) {                                              ③
                if (exclusive_console && con != exclusive_console)
                        continue;
                if (!(con->flags & CON_ENABLED))
                        continue;
                if (!con->write)
                        continue;
                if (!cpu_online(smp_processor_id()) &&
                    !(con->flags & CON_ANYTIME))
                        continue;
                con->write(con, text, len);
        }
}
```

①

#define console_loglevel (console_printk[0])

int console_printk[4] = {

```
    CONSOLE_LOGLEVEL_DEFAULT,     /* console_loglevel */
    MESSAGE_LOGLEVEL_DEFAULT,     /* default_message_loglevel */
    CONSOLE_LOGLEVEL_MIN,         /* minimum_console_loglevel */
    CONSOLE_LOGLEVEL_DEFAULT,     /* default_console_loglevel */
};
```

该array中的4个entry对应**/proc/sys/kernel/printk**中的4个值。

```
walterzh$ cat /proc/sys/kernel/printk
4    4    1    7
```

console_loglevel对应第一个值。

```
    if (level >= console_loglevel && !ignore_loglevel)
        return;
```

如果printk(LEVEL ...)中指定的LEVEL大于等于console_loglevel，则在串口中并不会输出。所以如果希望看到KERN_DEBUG level级别的log output,则需要如下动作：

```
# echo 8 4 1 7 > /proc/sys/kernel/printk
```

②

```
    if (!console_drivers)
        return;
```

```
struct console *console_drivers;
```

console_drivers组成一个struct console的linked-list。

凡是调用

void register_console(struct console *newcon) (in kernel/printk/printk.c)

的console都会链接到该linked-list上。

很多drivers/tty/serial directory下的串口driver都register console，这样该driver就可以作为kernel启动时的printk的输出通道。

在Gr2 / Gs2 LSP中serial driver pxa.c也调用了register_console()。

in drivers/tty/serial/pxa.c

serial_pxa_probe()

|

|

\|/

uart_add_one_port()

|

|

\|/

uart_configure_port()

|

|

\|/

register_console()

drivers/tty/vt/vt.c也注册了。

in drivers/tty/vt/vt.c

```
/*
 * This routine initializes console interrupts, and does nothing
 * else. If you want the screen to clear, call tty_write with
 * the appropriate escape-sequence.
 */

static int __init con_init(void)
{
        const char *display_desc = NULL;
        struct vc_data *vc;
        unsigned int currcons = 0, i;

        console_lock();

        if (conswitchp)
                display_desc = conswitchp->con_startup();
        if (!display_desc) {
                fg_console = 0;
                console_unlock();
                return 0;
        }

        for (i = 0; i < MAX_NR_CON_DRIVER; i++) {
                struct con_driver *con_driver = &registered_con_driver[i];

                if (con_driver->con == NULL) {
                        con_driver->con = conswitchp;
                        con_driver->desc = display_desc;
                        con_driver->flag = CON_DRIVER_FLAG_INIT;
                        con_driver->first = 0;
                        con_driver->last = MAX_NR_CONSOLES - 1;
                        break;
                }
        }

```

```
36.          for (i = 0; i < MAX_NR_CONSOLES; i++)
37.                  con_driver_map[i] = conswitchp;
38.
39.          if (blankinterval) {
40.                  blank_state = blank_normal_wait;
41.                  mod_timer(&console_timer, jiffies + (blankinterval * HZ));
42.          }
43.
44.          for (currcons = 0; currcons < MIN_NR_CONSOLES; currcons++) {
45.                  vc_cons[currcons].d = vc = kzalloc(sizeof(struct vc_data), GFP_NOWAIT);
46.                  INIT_WORK(&vc_cons[currcons].SAK_work, vc_SAK);
47.                  tty_port_init(&vc->port);
48.                  visual_init(vc, currcons, 1);
49.                  vc->vc_screenbuf = kzalloc(vc->vc_screenbuf_size, GFP_NOWAIT);
50.                  vc_init(vc, vc->vc_rows, vc->vc_cols,
51.                          currcons || !vc->vc_sw->con_save_screen);
52.          }
53.          currcons = fg_console = 0;
54.          master_display_fg = vc = vc_cons[currcons].d;
55.          set_origin(vc);
56.          save_screen(vc);
57.          gotoxy(vc, vc->vc_x, vc->vc_y);
58.          csi_J(vc, 0);
59.          update_screen(vc);
60.          pr_info("Console: %s %s %dx%d\n",
61.                  vc->vc_can_do_color ? "colour" : "mono",
62.                  display_desc, vc->vc_cols, vc->vc_rows);
63.          printable = 1;
64.
65.          console_unlock();
66.
67.  #ifdef CONFIG_VT_CONSOLE
68.          register_console(&vt_console_driver);
69.  #endif
70.          return 0;
71.  }
```

```
72.  console_initcall(con_init);
```

这样在start_kernel()调用console_init()时，会调用上面的con_init()。

```
1.  static struct console vt_console_driver = {
2.          .name           = "tty",
3.          .write          = vt_console_print,
4.          .device         = vt_console_device,
5.          .unblank        = unblank_screen,
6.          .flags          = CON_PRINTBUFFER,
7.          .index          = -1,
8.  };
```

vt_console_driver console会被链接到console_drivers linked-list上。

③

对linked-list上的console进行enumerate，但只有exclusive_console才允许作为printk的输出通道。

在register_console()中有如下code:

```
if (newcon->flags & CON_PRINTBUFFER) {
    /*
     * console_unlock(); will print out the buffered messages
     * for us.
     */
    raw_spin_lock_irqsave(&logbuf_lock, flags);
    console_seq = syslog_seq;
    console_idx = syslog_idx;
    console_prev = syslog_prev;
    raw_spin_unlock_irqrestore(&logbuf_lock, flags);
    /*
```

```
         * We're about to replay the log buffer.  Only do this to the
         * just-registered console to avoid excessive message spam to
         * the already-registered consoles.
         */
        exclusive_console = newcon;
    }
```

只有设置了CON_PRINTBUFFER flag的console才可以作为printk的输出console。而上面的vt_console_driver structure就设置了该flag。

所以当printk()真正输出时，调用的是drivers/tty/vt/vt.c的

void vt_console_print(struct console *co, const char *b, unsigned count)。

???

对ARM arch而言

在CONFIG_EARLY_PRINTK enable的情况下有early_printk console(Gr2 / Gs2 LSP enable CONFIG_EARLY_PRINTK)

in arch/arm/kernel/early_printk.c

```
1.  static void early_write(const char *s, unsigned n)
2.  {
3.          while (n-- > 0) {
4.                  if (*s == '\n')
5.                          printch('\r');
6.                  printch(*s);
7.                  s++;
8.          }
9.  }
10.
11. static void early_console_write(struct console *con, const char *s, unsigned n)
12. {
13.         early_write(s, n);
14. }
15.
16. static struct console early_console_dev = {
17.         .name =         "earlycon",
18.         .write =        early_console_write,
19.         .flags =CON_PRINTBUFFER| CON_BOOT,
20.         .index =        -1,
21. };
22.
23. static int __init setup_early_printk(char *buf)
24. {
25.         early_console = &early_console_dev;
26.         register_console(&early_console_dev);
27.         return 0;
28. }
29.
30. early_param("earlyprintk", setup_early_printk);
```

如果在kernel parameter "earlyprintk" enable，则printk()也会通过early_console_write() output。

early_console_write()其实就等同与arch/arm/kernel/setup.c中的early_print()，就是直接写初始化好的串口。

early_console_write()能work的条件同early_print()一样！是在觉得没有这个必要提供该功能。

in Documentation/kernel-parameters.txt

```
earlyprintk=    [X86,SH,BLACKFIN,ARM,M68k]
                earlyprintk=vga
                earlyprintk=efi
                earlyprintk=xen
                earlyprintk=serial[,ttySn[,baudrate]]
                earlyprintk=serial[,0x...[,baudrate]]
                earlyprintk=ttySn[,baudrate]
                earlyprintk=dbgp[debugController#]

                earlyprintk is useful when the kernel crashes before
                the normal console is initialized. It is not enabled by
                default because it has some cosmetic problems.

                Append ",keep" to not disable it when the real console
                takes over.

                Only one of vga, efi, serial, or usb debug port can
                be used at a time.

                Currently only ttyS0 and ttyS1 may be specified by
                name.  Other I/O ports may be explicitly specified
                on some architectures (x86 and arm at least) by
                replacing ttySn with an I/O port address, like this:
                        earlyprintk=serial,0x1008,115200
```

You can find the port for a given device in

/proc/tty/driver/serial:

2: uart:ST16650V2 port:00001008 irq:18 ...

Interaction with the standard serial driver is not

very good.

The VGA and EFI output is eventually overwritten by

the real console.

The xen output can only be used by Xen PV guests.

==========

另外在enable CONFIG_SERIAL_EARLYCON的情况下(Gr2 / Gs2 LSP没有enable该config value)。

in drivers/tty/serial/earlycon.c

也注册了printk()用到的console。

early_print()需要developer手工去hardcode的配置输出的UART通道（见《early_print support》笔记），而earlycon.c则把这种对UART的配置移到了dts中。（应该是更符合当前潮流）

in drivers/of/fdt.c

```
int __init early_init_dt_scan_chosen_serial(void)
{
        int offset;
        const char *p;
        int l;
        const struct of_device_id *match = __earlycon_of_table;
        const void *fdt = initial_boot_params;

        offset = fdt_path_offset(fdt, "/chosen");
        if (offset < 0)
                offset = fdt_path_offset(fdt, "/chosen@0");
        if (offset < 0)
                return -ENOENT;

        p = fdt_getprop(fdt, offset, "stdout-path", &l);
        if (!p)
                p = fdt_getprop(fdt, offset, "linux,stdout-path", &l);
        if (!p || !l)
                return -ENOENT;

        /* Get the node specified by stdout-path */
        offset = fdt_path_offset(fdt, p);
        if (offset < 0)
                return -ENODEV;

        while (match->compatible[0]) {
                unsigned long addr;
                if (fdt_node_check_compatible(fdt, offset, match->compatible)) {
                        match++;
                        continue;
                }

                addr = fdt_translate_address(fdt, offset);
                if (!addr)
                        return -ENXIO;
```

```
36.
37.                 of_setup_earlycon(addr, match->data);
38.                 return 0;
39.         }
40.         return -ENODEV;
41. }
42.
43. static int __init setup_of_earlycon(char *buf)
44. {
45.         if (buf)
46.                 return 0;
47.
48.         return early_init_dt_scan_chosen_serial();
49. }
50. early_param("earlycon", setup_of_earlycon);
```

在boot command上添加"earlycon" kernel parameter。

in Documentation/kernel-parameters.txt

earlycon= [KNL] Output early console device and options.

cdns,<addr>

Start an early, polled-mode console on a cadence serial port at the specified address. The cadence serial port must already be setup and configured. Options are not yet supported.

uart[8250],io,<addr>[,options]

uart[8250],mmio,<addr>[,options]

uart[8250],mmio32,<addr>[,options]

Start an early, polled-mode console on the 8250/16550

UART at the specified I/O port or MMIO address.

MMIO inter-register address stride is either 8-bit

(mmio) or 32-bit (mmio32).

The options are the same as for ttyS, above.

pl011,<addr>

Start an early, polled-mode console on a pl011 serial

port at the specified address. The pl011 serial port

must already be setup and configured. Options are not

yet supported.

msm_serial,<addr>

Start an early, polled-mode console on an msm serial

port at the specified address. The serial port

must already be setup and configured. Options are not

yet supported.

msm_serial_dm,<addr>

Start an early, polled-mode console on an msm serial

dm port at the specified address. The serial port

must already be setup and configured. Options are not

yet supported.

smh Use ARM semihosting calls for early console.

在dts中可以配置输出的UART信息，比如

chosen {

```
        name = "chosen";
        bootargs = "root=/dev/ram rw console=ttyS0,115200";
        linux,stdout-path = "/soc8349@e0000000/serial@4500";
};
```

这里soc8349@e0000000是UART device node。

==========

在Gr2 / Gs2 LSP中依次注册了3个console。

1. register_console-earlycon (arch/arm/kernel/early_printk.c)
2. register_console-tty (drivers/tty/vt/vt.c)
3. register_console-ttyS (pxa.c UART)

(1)在处理early_param之时

(2)在start_kernel()的console_init()之时

(3)在start_kernel()的后期，各个driver初始化之时

==========

当pxa.c UART driver初始化完成以后，console会发生切换，如下：

```
d4030000.uart: ttyS0 at MMIO 0xd4030000 (irq = 48, base_baud = 691182) is a UART1

walter: register_console-ttyS

console [ttyS0] enabled
console [ttyS0] enabled
```

walter: unregister_console-earlycon

walter: unregister_console-earlycon

bootconsole [earlycon0] disabled

bootconsole [earlycon0] disabled

==========

在start_kernel()中dump出console_drivers linked-list。

```
        /*
         * HACK ALERT! This is early. We're enabling the console before
         * we've done PCI setups etc, and console_init() must be aware of
         * this. But we do want output early, in case something goes wrong.
         */
        console_init();
        if (panic_later)
                panic("Too many boot %s vars at `%s'", panic_later,
                      panic_param);

        lockdep_info();

        extern struct console *console_drivers;

        printk("walter: dump console\n");
        struct console *next = console_drivers;
        while(next)
        {
                printk("walter: %s\n", next->name);
                next = next->next;
        }
```

输出如下：

walter: dump console

walter: earlycon

也就是说在Gr2 / Gs2 LSP中printk用的是arch/arm/kernel/early_printk.c中early_console_write()。