Consolations for Kernel Hackers

BY TOM JONES

You turn the machine on, it buzzes and whirrs, and there is a chime. The display jumps to a bright, but black screen, a first sign that the sounds aren't just distant movements. Some text from Phoenix or American scrolls by, and then you are shown the wonder of the FreeBSD boot prompt. 10 seconds later, the countdown and text begin scrolling. For some, this is a primitive machine, but for many of us, this is the best part. Being close to the metal at the start.

Kernel messages start to scroll by, beginning with the project copyright and continuing with devices as they are discovered and attached, until the kernel hands over to userspace. The text changes in aspect to indicate this, and we start seeing services start. All kernel hackers have sat watching this text scroll by, looking for the message they added to a new driver or an old piece of code that isn't behaving as we all thought it would. We are waiting for our `printf` to be called, proving that we have built and run a kernel with our changes, that we have finally tamed the build process, and can get to real work.

All developers have used `printf` debugging and print statements to track execution flow, figure out how or if things are working, and where they connect. Most of the time, this core part of the kernel functions flawlessly; we get text output, or the kernel has died before it can be shown. This cinema is backed by a console driver, a piece of code that sits right at the bottom of `printf` at the interface between hardware and software. If you are embroiled in working on a console driver or in a very early boot, you want to know how the console works normally, so you can make it work abnormally and get some debug information you might not be able to otherwise.

The machines we use now are quite different from those that UNIX was designed around.

Terminals and Consoles

The machines we use now are quite different from those that UNIX was designed around, which means that the boundary between the system console, an access terminal, and a graphical output isn't very clear. In the old days, the system would come up with a dedicated system console, something that was in or near the machine room where the computer lived. Terminals could live somewhere else quieter, either in a terminal room or on some lucky person's desk.

The console, though, was the heart of the machine; if the system needed to tell an operator about an exciting or dangerous failure, it would pop up on the console. The console is a core component of the system; most of the time, it is chatting away, printing messages on graphics output, which are unconnected or out to a serial port with no cable connected.

Most of the time, these messages go unused. They are the startup text the kernel outputs as it finds, discovers, and configures devices. While operating, these messages indicate hardware and system errors and notify us of buffer overflows and exhausted limits. When we need to see the messages, we can pull the buffer with the `dmesg` command. Usually, that is all anyone really needs.

But kernel hackers are an odd bunch, and sadly, most of the time, `printf` debugging remains the best way to track down errors and get signs of life from a new driver. If you get distracted and start digging into how exactly `printf` works in the kernel, you eventually start unearthing `cn*` functions and the console drivers' dispatch table.

Walking Down from `printf`

For most developers, kernel code is low-level, but our highest point today is any piece of code that can call `printf(9)`. Let's walk down from `printf` through some of the layers to get a sense of what's happening before a console driver is called. `printf(9)` lets us see what is usually the simplest case. `printf(9)` and its friends are implemented in `sys/kern/subr_prf.c`; the functions are mostly wrappers around `_vprintf`, which also handles outputting any buffered messages and putting bytes onto console devices via `kvprintf`.

```
static int
_vprintf(int level, int flags, const char *fmt, va_list ap)
{
    struct putchar_arg pca;
    int retval;
#ifdef PRINTF_BUFR_SIZE
    char bufr[PRINTF_BUFR_SIZE];
#endif
    TSENTER();
    pca.tty = NULL;
    pca.pri = level;
    pca.flags = flags;
#ifdef PRINTF_BUFR_SIZE
    pca.p_bufr = bufr;
    pca.p_next = pca.p_bufr;
    pca.n_bufr = sizeof(bufr);
    pca.remain = sizeof(bufr);
    *pca.p_next = '\0';
#else
    /* Don't buffer console output. */
    pca.p_bufr = NULL;
#endif
    retval = kvprintf(fmt, putchar, &pca, 10, ap);
    #ifdef PRINTF_BUFR_SIZE
    /* Write any buffered console/log output: */
    if (*pca.p_bufr != '\0')
          prf_putbuf(pca.p_bufr, flags, level);
#endif
    TSEXIT();
    return (retval);
}
```

It parses the format string, which resolves to a call to **putchar()**. **putchar** calls **cnput_char**, and finally we call into the driver dispatch table for driver-specific calls. **kvprintf** takes a function pointer to use for the actual outputting of bytes.

```
int
kvprintf(char const *fmt, void (*func)(int, void*), void *arg, int radix, va_list ap)
{
#define PCHAR(c) {int cc=(c); if (func) (*func)(cc,arg); else *d++ = cc; retval++; }
     char nbuf[MAXNBUF];
     char *d;
     const char *p, *percent, *q;
     u_char *up;
     int ch, n, sign;
     uintmax_t num;
```

In our example above, the function point is to **putchar**

```
/*
 * Print a character on console or users terminal. If destination is
 * the console then the last bunch of characters are saved in msgbuf for
 * inspection later.
 */
static void
putchar(int c, void *arg)
{
struct putchar_arg *ap = (struct putchar_arg*) arg;
struct tty *tp = ap->tty;
int flags = ap->flags;

/* Don't use the tty code after a panic or while in ddb. */
if (kdb_active) {
      if (c != '\0')
            cnputc(c);
      return;
}
if ((flags & TOTTY) && tp != NULL && !KERNEL_PANICKED())
       tty_putchar(tp, c);
if ((flags & TOCONS) && cn_mute) {
       flags &= ~TOCONS;
       ap->flags = flags;
      }
      if ((flags & (TOCONS | TOLOG)) && c != '\0')
            putbuf(c, ap);
}
```

cnputc lives at the bottom of the kernel with many paths to it.

```
  void
  cnputc(int c)
  {
      struct cn_device *cnd;
      struct consdev *cn;
```

```
        const char *cp;
        if (cn_mute || c == '\0')
                return;
    (2)
    #ifdef EARLY_PRINTF
         if (early_putc != NULL) {
                if (c == '\n')
                        early_putc('\r');
                early_putc(c);
                return;
         }
    #endif
      (1)    STAILQ_FOREACH(cnd, &cn_devlist, cnd_next) {
                cn = cnd->cnd_cn;
                if (!kdb_active || !(cn->cn_flags & CN_FLAG_NODEBUG)) {
                        if (c == ‘\n’)
                                cn->cn_ops->cn_putc(cn, '\r');
                        cn->cn_ops->cn_putc(cn, c);
                }
        }
        if (console_pausing && c == ‘\n’ && !kdb_active) {
                for (cp = console_pausestr; *cp != '\0'; cp++)
                        cnputc(*cp);
                cngrab();
                if (cngetc() == '.')
                        console_pausing = false;
                cnungrab();
                cnputc('\r');
                for (cp = console_pausestr; *cp != '\0'; cp++)
                        cnputc(' ');
                cnputc('\r');
        }
}
```

cnputc has two important pieces of code, the code in the loop at (1) walks a list of attached consoles and calls the consoles **cn_putc** callback. For reasons related to locking in and the debugger, it doesn't call these methods if the kernel debugger **kdb** is active.

This is where FreeBSD implements support for multiple consoles. If you ever wondered how, you get a glass terminal on a graphical display and a serial port, that's where it happens. The code at (2), wrapped in **#ifdef EARLY_PRINTF**, is incredibly platform-specific, enabling output from the first bytes of the kernel image. If defined and supported on your hardware, you can retrieve bytes from the loader handover.

As an example of how platform-specific **early_putc** is, here is the **x86/amd64** implementation.

```
#if CHECK_EARLY_PRINTF(ns8250)
#if !(defined(__amd64__) || defined(__i386__))
#error ns8250 early putc is x86 specific as it uses inb/outb
#endif
static void
```

```
uart_ns8250_early_putc(int c)
{
     u_int stat = UART_NS8250_EARLY_PORT + REG_LSR;
     u_int tx = UART_NS8250_EARLY_PORT + REG_DATA;
     int limit = 10000; /* 10ms is plenty of time */
     while ((inb(stat) & LSR_THRE) == 0 && --limit > 0)
          continue;
     outb(tx, c);
}
early_putc_t *early_putc = uart_ns8250_early_putc;
#endif /* EARLY_PRINTF */
```

This **early_putc** uses the x86 **outb** instruction to transmit bytes on a serial port, but it has never been given support for working with MMIO (memory mapped IO) UARTs which are now quite common.

(As I was writing this article, MMIO UART support on arm64 was added.)

Console Life Cycle

To look at the console life cycle, it is best to start from the FreeBSD Loader. Booting FreeBSD requires two Operating System components, the kernel and our boot loader (**loader(8)**). All hardware platforms start with some sort of firmware. In old x86 systems, this was the bios looking for a boot block. Modern x86 systems start via firmware, which presents the EFI interface. Non-PC systems boot in different ways. In embedded devices, a boot ROM on the system-on-a-chip tests and tries different boot media; the first one that works is used. On a device, that might mean that the boot ROM tries in order:

- NAND flash
- SD Card
- USB media
- A USB firmware mode

In many cases, the boot firmware will document what it is trying over that good old reliable serial interface. Typically, on ARM and RISC-V systems, a second-stage bootloader, called a secondary program loader, or SPL, is run. Almost always for us, this is u-boot. Early firmware will only print to a serial port, but u-boot in many cases has enough drivers together to be able to use a framebuffer device if possible.

Loader is modular and runs in many different forms depending on the base platform; we run from u-boot, as an EFI application, on top of open firmware, and in other places. For bhyve virtual machine images, the FreeBSD loader runs as a userspace application. Most of these platforms use a static console as the default output method and can, if possible, be configured to use additional outputs.

FreeBSD's loader runs as a chained application or a sub-application on top of the firmware or u-boot. U-boot will run EFI applications, making it much easier to create usable FreeBSD boot media. Without any other configuration, FreeBSD's loader determines what to use as the system console (the place where messages go). On an EFI system, we will use the EFI console, which has the unfortunate property of sometimes using buggy EFI interfaces to output characters. If you have ever used FreeBSD loader on a serial port on an EFI system, you might have seen duplicated bytes on the serial port — this is due to the EFI output routine being used in the usual EFI cons output and the EFI serial console output.

Loader can be configured to use multiple consoles via the config file or from the command line by adding more consoles to the **consoles** environment variable.

```
> set console="efi comconsole"
```

Loader passes configuration information to the kernel in the kernel image, along with some variables. The console type and whether multiple consoles are used are controlled by the **boothow** variable, which is set to an **RB_*** value. Historically (all the world's a VAX!), these were the system call arguments to **reboot(9)**, but they have been reused and become intertwined over time. From loader, we frequently get **RB_SERIAL** (use a serial port), **RB_MULTIPLE** (use multiple consoles), **RB_SINGLE** (boot to single-user), and **RB_VERBOSE**, which sets bootverbose in the kernel and fills up your message buffer.

Starting the Kernel

Eventually loader will start the selected kernel and pass along the arguments and environment. Everything that loader has done has been temporary, and all the work has to be done again in the kernel to get a working system that the kernel is in control of. Console initialisation is tied closely to platform-specific hardware. Each FreeBSD platform has its own machine-dependent code that runs and sets up memory and devices before the system can perform machine-independent work.

On most platforms this is an **init*** call (like **initarm**, **init386**, **initriscv** or the frustratingly unmatching **powerpc_init**), but on amd64 this is **hammer_time**. Each of these routines may try twice to initialise consoles (currently only i386 and amd64 do) based on configuration from loader. The relevant part of **hammer_time** looks like this:

```
/*
 * The console and kdb should be initialized even earlier than here,
 * but some console drivers don't work until after getmemsize().
 * Default to late console initialization to support these drivers.
 * This loses mainly printf()s in getmemsize() and early debugging.
 */
 TUNABLE_INT_FETCH("debug.late_console", &late_console);
 if (!late_console) {
        cninit();
        amd64_kdb_init();
 }

 getmemsize(physfree);
 init_param2(physmem);

     /* now running on new page tables, configured,and u/iom is accessible */

#ifdef DEV_PCI
     /* This call might adjust phys_avail[]. */
      pci_early_quirks();
#endif

     if (late_console)
            cninit();
```

You can see from the example that two attempts to call **`cninit`** are possible, depending on the value of the **`late_console`** variable, which defaults to true. Sandwiched between these two calls is the discovery and allocation of system memory. Before one of these calls to **`cninit`** it is not possible to output anything to the system console (it doesn't exist!), any **`printf`** before **`cninit`** will evaporate into the ether.

The exception is if **`eary_putc`** is defined as described, then you can do some early debugging. **`cninit`** walks a statically defined list of consoles and tries to find the best possible console.

```
/*
 * Find the first console with the highest priority.
 */
best_cn = NULL;
SET_FOREACH(list, cons_set) {
    cn = *list;
    cnremove(cn);
    /* Skip cons_consdev. */
    if (cn->cn_ops == NULL)
          continue;
    cn->cn_ops->cn_probe(cn);
    if (cn->cn_pri == CN_DEAD)
          continue;
    if (best_cn == NULL || cn->cn_pri > best_cn->cn_pri)
          best_cn = cn;
    if (boothowto & RB_MULTIPLE) {
          /*
           * Initialize console, and attach to it.
           */
          cn->cn_ops->cn_init(cn);
          cnadd(cn);
    }
}
if (best_cn == NULL)
      return;
if ((boothowto & RB_MULTIPLE) == 0) {
      best_cn->cn_ops->cn_init(best_cn);
      cnadd(best_cn);
}
if (boothowto & RB_PAUSE)
      console_pausing = true;
     /*
      * Make the best console the preferred console.
      */
     cnselect(best_cn);
#ifdef EARLY_PRINTF
     /*
      * Release early console.
      */
     early_putc = NULL;
#endif
```

Consoles are sorted by a priority set by the console driver when its **cn_probe** function is called. At the end, we must have a console (there is always a default), and we release **early_putc** so that it is gone forever. I didn't show the matching code in **loader.efi**, but it is very similar. The important exception for anyone hacking on EFI consoles is the lack of an **early_putc** mechanism to give you some last chance hope at practical **printf** debugging. However, this can be hacked around. In the EFI environment, the output routine is available, and if you just hang on to one of the consoles while doing the **init**, you can get more visibility into what is happening.

In the EFI environment, the output routine is available.

Available Consoles

The system consoles are defined at kernel build time. In **cninit** above, we walk through **cons_set** — a linker set generated for us by the build process. Each console is not declared as a normal device driver; instead, it is defined with the **CONSOLE_DRIVER** macro, which adds the console to the linker set. As of 16-CURRENT, linker sets aren't populated on kernel module load, meaning you cannot dynamically add console drivers to the kernel at run time.

The **CONSOLE_DRIVER** macro takes the name of the console to add to the linker set and handles the creation of all the callbacks based on the name:

```
CONSOLE_DRIVER lives in sys/sys/cons.h
#define CONSOLE_DEVICE(name, ops, arg)                                  \
        static struct consdev name = {                          \
                .cn_ops = &ops,                         \
                .cn_arg = (arg),                        \
        };                                      \
        DATA_SET(cons_set, name)
#define CONSOLE_DRIVER(name, ...)                                \
        static const struct consdev_ops name##_consdev_ops = {          \
                /* Mandatory methods. */                \
                .cn_probe = name##_cnprobe,                \
                .cn_init = name##_cninit,                \
                .cn_term = name##_cnterm,                \
                .cn_getc = name##_cngetc,                \
                .cn_putc = name##_cnputc,                \
                .cn_grab = name##_cngrab,                \
                .cn_ungrab = name##_cnungrab,                \
                /* Optional fields. */      \
                __VA_ARGS__                         \
        };                                      \
        CONSOLE_DEVICE(name##_consdev, name##_consdev_ops, NULL)
```

The macro defines the mandatory callbacks that a console driver must implement and provides space to add extra code. Currently, **uart_tty** takes advantage of this to add a resume method. The console callbacks are all quite minimal; we really want the console to work after all. **cn_probe** needs to set the console priority (**cn_pri**) to something other than **CN_DEAD** for the console driver to be considered by **cninit**.

`cn_init` is called for selected console devices by `cninit` (either the best one or all non-dead consoles if we are booting with `RB_MULTIPLE`). In `cn_init` the console driver can establish state, as an example `uart_cninit` stores a cookie so the console isn't initialised twice. `cn_term` is called if the console driver is ever removed to tidy things up. `cn_grab` and `cn_ungrab` are used to disable and enable interrupts if the kernel is going to print a series of characters (or read one in). Finally, `cn_putc` and `cn_get` handle output and input for the console device. The magic we care about.

Outputting Bytes

There are many different devices that can be used as a system console. It is sometimes hard to separate the pc from the x86 (or AMD64) platform, but a big part of what makes it a platform rather than just a processor architecture is the ecosystem of peripherals and hardware that can be expected to be present. On the PC we can expect to see 1 or more UARTs (serial ports). This is so fundamental that FreeBSD as of 15.0 still assumes there will be two and statically configures them. UARTs were commonly hooked out to explicit instructions (`inb` and `outb`), which allowed a caller to send data out the serial port with a single instruction.

This simple interface is deliberate; if you are bringing up a machine from nothing, it is very handy to be able to get debug information from the system with a minimal proof of concept.

This simple interface allows a program (or a driver) to interact with the outside world. If you are the operator of a new machine and want to know what the system is up to, then connecting a serial printer, a teletype, or, in the year 2026, a terminal program and serial cable let you get debug information from even the smallest code examples.

Instructions for input and output of bytes on a serial port put limitations on processor and hardware designers. You can't expect an instruction for every device, and so we have moved mostly to memory-mapped peripherals. The same 8250 and 16650 devices used in the original PC are available in modern systems, with compatible hardware blocks in modern system-on-chip designs. Rather than using `inb` and `outb`, they are memory-mapped peripherals.

There are many different devices that can be used as a system console.

Other systems expose slightly different devices for their console, but on almost every platform, this boils down to using the old National Semiconductor 8250 or 16650 UART interface for getting bytes onto the wire. This means we see platform-specific console devices, but many, many of them resolve down to the same battle-tested code.

I don't think anyone who has used a computer in the last 3 decades would be very happy with just a serial port. From the beginning, the pc supported graphical output, and the BIOS made accessing it practical with very simple early program loaders. With a serial port and a graphical output, it doesn't seem right to explicitly bake in `outb` into `printf` for getting bytes out of a serial port. So, while a very simple console driver is possible, we end up building a console driver subsystem that lets us reuse much of the code while still allowing platform and hardware-specific variations.

Loader

So far, we have mentioned loader quite a few times as it was relevant during kernel start-up, but it is an operating system in its own right. Loader prepares the kernel's operating environment and configures the root volume, which provides the userspace environment for kernel modules. All platforms that FreeBSD runs on have a loader of some sort (well, we can directly boot the kernel in some cases, but that leaves much functionality on the table).

Loader is started either by the platform firmware or a secondary loader. In most deployments, the loader runs in the EFI environment (on arm64, arm32, amd64, and riscv). This environment provides a lot of early services to loader, making its job easier and portability more straightforward. As loader runs in many environments and supports 2 scripting environments (Lua and Forth, yes, Forth in 2026), it provides some of its own abstractions to make core services easier to use. Loader also has console drivers, and they look very similar to the kernel ones.

```
stand/common/bootstrap.h:
  /*
   * Modular console support.
   */
  struct console
  {
       const char      *c_name;
       const char      *c_desc;
       int             c_flags;
#define C_PRESENTIN     (1<<0)      /* console can provide input */
#define C_PRESENTOUT    (1<<1)      /* console can provide output */
#define C_ACTIVEIN      (1<<2)      /* user wants input from console */
#define C_ACTIVEOUT     (1<<3)      /* user wants output to console */
#define C_WIDEOUT       (1<<4)      /* c_out routine groks wide chars */
       /* set c_flags to match hardware */
       void    (* c_probe)(struct console *cp);
       /* reinit XXX may need more args */
       int             (* c_init)(int arg);
       /* emit c */
       void            (* c_out)(int c);
       /* wait for and return input */
       int             (* c_in)(void);
       /* return nonzero if input waiting */
       int             (* c_ready)(void);
  };
```

Loader console drivers have in and out methods, as the kernel ones do; an init method, a probe method, and a ready method to speed up polling for input. Rather than priorities, loader console drivers have a flags argument. Each variant of loader has its own **`conf.c`** file, which defines its platform-specific features and implementations. One of these features is a list of consoles that can be used on that platform. For **`loader.efi`** we have:

```
#if defined(__amd64__) || defined(__i386__)
extern struct console comconsole;
extern struct console nullconsole;
```

```
extern struct console spinconsole;
#endif
struct console *consoles[] = {
     &efi_console,
     &eficom,
#if defined(__aarch64__) && __FreeBSD_version < 1500000
     &comconsole,
#endif
#if defined(__amd64__) || defined(__i386__)
     &comconsole,
     &nullconsole,
     &spinconsole,
#endif
     NULL
};
```

For an amd64 machine our full consoles list is:

- **efi_console**: Console using the efi output routines
- **eficom**: Console using the efi serial output routines. This one can be a little strange and lead to duplicated output as the EFI implementation uses the framebuffer and the serial port.
- **comconsole**: Console on a traditional COM port, a serial console using the
- **nullconsole**: All output bytes disappear into the void.
- **spinconsole**: Each output byte progresses a spinner by one step; this gives a visual indication of something happening, but no way to debug or provide input.

There are platform-specific consoles for the userboot loader (this is for start bhyve instances), the kboot (boot from Linux), u-boot, and other platforms. On amd64 **comconsole** and **eficom** map to the same driver functions.

EFI Loader Console init

The load **cons_probe** function handles discovering consoles and configuring them. It is called very early in **efi/loader/main.c**:

```
EFI_STATUS
main(int argc, CHAR16 *argv[])
{
...
#if !defined(__arm__)
      efi_smbios_detect();
#endif

      /* Get our loaded image protocol interface structure. */
      (void) OpenProtocolByHandle(IH, &imgid, (void **)&boot_img);

      /* Report the RSDP early. */
      acpi_detect();

      /*
       * Chicken-and-egg problem; we want to have console output early, but
       * some console attributes may depend on reading from eg. the boot
```

```
         * device, which we can't do yet.  We can use printf() etc. once this is
         * done. So, we set it to the efi console, then call console init. This
         * gets us printf early, but also primes the pump for all future console
         * changes to take effect, regardless of where they come from.
         */
        setenv("console", "efi", 1);
        uhowto = parse_uefi_con_out();

#if defined(__riscv)
        /*
         * This workaround likely is papering over a real issue
         */
        if ((uhowto & RB_SERIAL) != 0)
                setenv("console", "comconsole", 1);
#endif
        cons_probe();
        /* Set print_delay variable to have hooks in place. */
        env_setenv("print_delay", EV_VOLATILE, "", setprint_delay, env_nounset);

        /* Set up currdev variable to have hooks in place. */
        env_setenv("currdev", EV_VOLATILE, "", gen_setcurrdev, env_nounset);

        /* Init the time source */
        efi_time_init();
```

The chicken-and-egg comment is common in the tree. Basically, we need to be able to debug what is happening as early as possible, and a console is the easiest way to set that up. The EFI environment we have right now can output strings, but we don't have console infrastructure. We play a trick on all platforms and statically pick a console before looking for consoles.

Console probing happens before we can read from storage, so it starts with assumptions and then reassesses. On EFI we can output data before **`cons_probes`** by making a direct EFI with **`ST->ConOut->OutputString`**. We have console infrastructure, and that lets us use **`printf`**, so we should. **`cons_probe`** walks the list and finds a working console. The selected console is added to the **`console`** environment variable in loader and can be changed or added to by environment variables from **`loader.conf`** or on the command line.

Once consoles are selected, they are plummed into **`printf`** in a similar way to kernel consoles.

```
common/console.c:
  void
  putchar(int c)
  {
        int     cons;
        /* Expand newlines */
        if (c == '\n')
                putchar('\r');
        for (cons = 0; consoles[cons] != NULL; cons++) {
                if ((consoles[cons]->c_flags & (C_PRESENTOUT | C_ACTIVEOUT)) ==
```

```
                (C_PRESENTOUT | C_ACTIVEOUT))
                  consoles[cons]->c_out(c);
        }
        /* Pause after printing newline character if a print delay is set */
        if (print_delay_usec != 0 && c == '\n')
        delay(print_delay_usec);
}
```

Debugging a Console Driver

So, there is all the magic in a console driver, and on reflection, it is a bit of a dim light when we look at it in detail. **`printf`** is quite simple, and it is driving a single character output routine. It is similar between the kernel and loader, I know you are wondering why I repeated the dive with loader, but this was to show that it is the same and that debugging tricks that work in the kernel will probably work in loader environment. If you are stuck pre-console or have an issue that perturbs the working console, you will reach for anything you can to debug what is happening.

So how can we debug issues with console drivers?

The first and best two suggestions I have are to use a debugger or a second console. If you can use a debugger for loader or the early boot environment, then you really should; it will cut through all the rest of the stuff. You may be able to use a debugger if you have a second working console (the kernel debugger leverages the console infrastructure) or if you can control the machine under development holistically. Either using an emulator such as **`qemu`** or with a hardware debug interface like JTAG. Setting these up is beyond the scope of this article, and I know that if you've read this far, you are thinking, "But I'm only doing this thing because I can't debug!"

A second console is a great option for when you need to debug the console, but it probably won't help so much early in the kernel. If you have a working console and questions about the non-working console, you can bypass much of the infrastructure and either configure things you know will be there statically or directly hook into the console functions you want to use.

The first and best two suggestions I have are to use a debugger or a second console.

This is where the linker set for the list of consoles and the static list in loader really helps. Just pull out the console you want from the list and take **`put_c`** and do what you need. Doing this skips all the locks and loses you **`printf`**, but it also skips all the locks, and it might let you get debugging information out.

I have used this direct capture to debug a loader console driver; loader doesn't have an **`early_putc`**, so hacks were needed. Usually, during **`cons_probe`** in loader, we walk all the consoles and remove the **`ACTIVE_OUT`** flag; this means that if we use **`printf`**, we get no data out. To debug console detection, I statically configured the first console (knowing it would be there and working) and used **`printf`** to debug the start-up of my driver of interest.

The change looked something like this:

```
diff --git a/stand/common/console.c b/stand/common/console.c
index 65ab7ffad622..97201a7479b3 100644
--- a/stand/common/console.c
+++ b/stand/common/console.c
@@ -107,14 +107,18 @@ cons_probe(void)
   env_setenv("twiddle_divisor", EV_VOLATILE, "16", twiddle_set,
     env_nounset);

+   /* capture the first console and keep it working while parsing others */
+    consoles[0]->c_flags |= C_ACTIVEIN | C_ACTIVEOUT;
+    consoles[0]->c_init(0);
+
    /* Do all console probes */
-    for (cons = 0; consoles[cons] != NULL; cons++) {
+    for (cons = 1; consoles[cons] != NULL; cons++) {
         consoles[cons]->c_flags = 0;
         consoles[cons]->c_probe(consoles[cons]);
     }
     /* Now find the first working one */
     active = -1;
-    for (cons = 0; consoles[cons] != NULL && active == -1; cons++) {
+    for (cons = 1; consoles[cons] != NULL && active == -1; cons++) {
         consoles[cons]->c_flags = 0;
         consoles[cons]->c_probe(consoles[cons]);
         if (consoles[cons]->c_flags == (C_PRESENTIN | C_PRESENTOUT))
```

The final hack I have with console drivers is to explicitly take control of the output function. **`early_putc`** is a good feature, but it only works for some devices and is dropped at **`cons_probe`**. If you pull out the **`cn_putc`** for a console driver you need to use or test, then you have an output method you can use sans **`printf`** when you need it.

Recently, trying to understand why a uart wouldn't output anything I did, added this crime to **`cons_probe`**:

```
if (uart == NULL || cp == NULL) {
      printf("uart is NULL\n");
} else {
    // this won't work if we boot multi, it'll panic on reinit
      printf("trying to init\n");
      uart->cn_ops->cn_init(uart);
      printf("trying to print\n");
      uart->cn_ops->cn_putc(uart, 'H');
      uart->cn_ops->cn_putc(uart, 'E');
      uart->cn_ops->cn_putc(uart, 'L');
      uart->cn_ops->cn_putc(uart, 'L');
      uart->cn_ops->cn_putc(uart, '\r');
      uart->cn_ops->cn_putc(uart, '\n');

#define TEST_COUNT 100
      printf("Tring to print %d 4's\n", TEST_COUNT);
      for (int i = 0; i < TEST_COUNT; i++)
```

```
            uart->cn_ops->cn_putc(uart, '4');

        uart->cn_ops->cn_putc(uart, '\r');
        uart->cn_ops->cn_putc(uart, '\n'ß);
}

for( int i = 0; i < 40; i++) {
        DELAY(100000);
}
```

We manually set up the uart and output bytes, I was trying to understand why no data was coming out of the port, and that loop outputting `4` gave me something to sync an oscilloscope to.

Consolations

If you have to debug a console driver or work in loader or the kernel in the pre, or very early boot, then you will take what you can get, to get data out of the system. We don't have to toggle a GPIO; thankfully, the console system is pretty simple in design and use. That means it should work, but if it doesn't, you can bend it to output data when you need it to. These sorts of intermediate hacks are horrible and never appear in the source tree, but they can be very useful when you need to debug something, and it won't comply.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.