The Linux Kernel's Interrupt Controller API

William Gatliff

Table of Contents

Overview

This paper describes the API Linux uses to interact with a host platform's interrupt controller hardware. This knowledge is essential when porting Linux to a custom platform: without a thorough understanding of how interrupts are incorporated into the Linux kernel, a functional embedded Linux system is not possible.

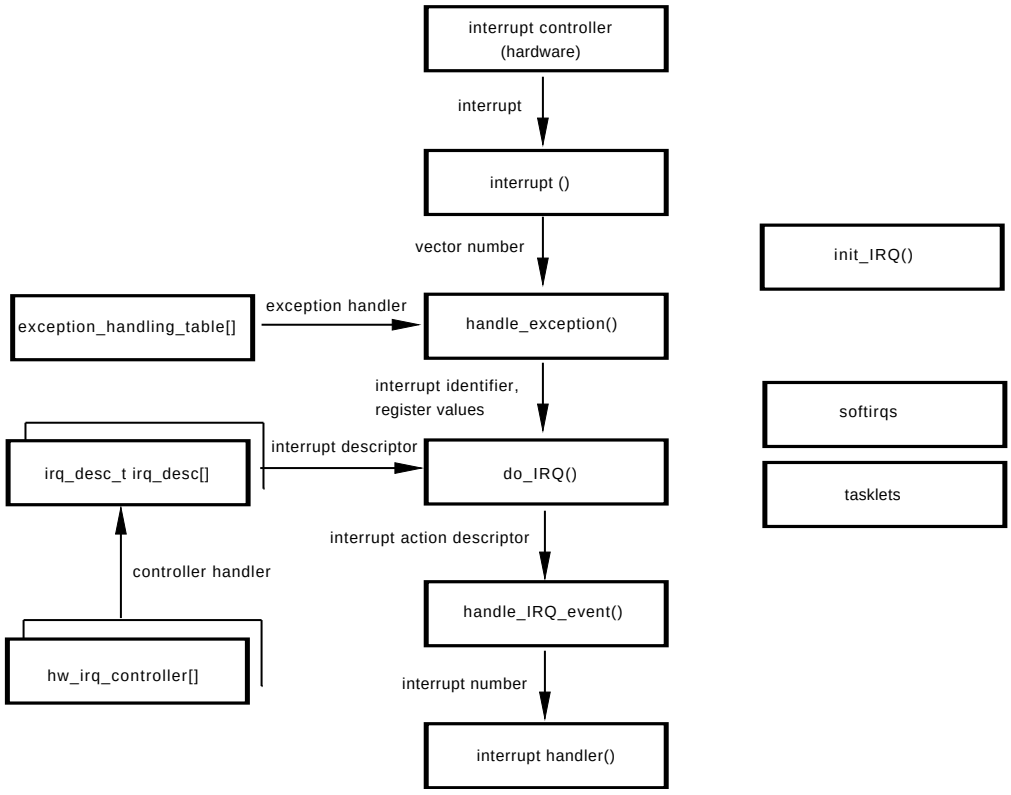
The presentation begins by describing the two most important data structures in the Linux kernel related to interrupt management, the irq_desc_t structure, and the hw_irq_controller structure. The discussion then moves to the `do_IRQ()`, `handle_IRQ_event()` and `handle_exception` functions, and concludes with a brief discussion of kernel initialization related to interrupt management, and how softirqs and tasklets relate to interrupt handling.

To make the descriptions clearer, this article includes source code taken from the Linux kernel that runs on the Sega Dreamcast gaming console. This platform utilizes a Hitachi SH-4 microprocessor, which is

well documented, embedded-friendly, and considerably less complicated than other chips commonly associated with Linux. The author is a member of the Linux Dreamcast development team, and has published several papers related to embedded Linux on that and other platforms.

A visual representation of Linux's interrupt handling system is shown in Figure 1. This figure depicts the relationships between various kernel structures and code, and forms the framework for the rest of the discussion.

Figure 1. Linux interrupts, a pictoral representation.



The irq_desc_t Structure

The foundation of Linux's interrupt management code is the irq_desc_t structure. An array of this structure, called `irq_desc[]`, keeps track of every interrupt request source in a Linux system. The definition for irq_desc_t is shown in Figure 2, which was taken from `include/linux/irq.h` in the kernel source code.

Figure 2. The irq_desc_t structure.

```
typedef struct {
  unsigned int status;
  hw_irq_controller *handler;
  struct irqaction *action;
  unsigned int depth;
  spinlock_t lock;
} irq_desc_t;
```

The *irq_desc_t.status* field

The *status* field records the status of the interrupt source. An interrupt source's status is a combination of one or more of the bitmaps shown in Table 1. Most of the bitmaps are mutually exclusive, and many of the values only have significance in SMP environments. They are still used in single-processor settings, however, to maintain consistency between kernel versions.

Table 1. Bitmaps for *irq_desc_t.status*.

IRQ_INPROGRESS	Interrupt is currently being handled.
IRQ_DISABLED	Interrupt source is disabled.
IRQ_PENDING	Interrupt needs handling.
IRQ_REPLAY	Interrupt has been replayed but not acknowledged yet. This is a hack to deal with buggy interrupt controllers that drop requests in SMP settings.
IRQ_AUTODETECT	Interrupt source is being probed.
IRQ_WAITING	Probing, waiting for interrupt.
IRQ_LEVEL	Interrupt source is level-triggered.
IRQ_MASKED	Interrupt source is masked, and should not be active.
IRQ_PER_CPU	Interrupt has been assigned to a particular CPU. Only used in SMP environments.

The *irq_desc_t.action* field

The action field is a linked list of irqaction structures, each of which records the address of an interrupt handler for the associated interrupt source. One irqaction structure is created and added to the end of the list for each call to the kernel's `request_irq()` function. The complete definition of the irqaction structure is shown in Figure 3.

Figure 3. The irqaction structure.

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

Interrupt handlers referenced by the *irqaction.handler* field are not microprocessor-level interrupt service routines, and therefore do not exit via `RTE` or similar interrupt-related opcodes. Instead, they are simply functions associated with the handling of interrupts from particular external devices, and they have minimal knowledge (if any) of the means by which those interrupt requests are delivered to the host microprocessor. This makes interrupt-driven device drivers largely portable across different microprocessor architectures.

The *irq_desc_t.depth* field

Some interrupt controllers don't like multiple or unbalanced enable/disable requests to an interrupt source. The *depth* field makes sure that enable and disable requests remain balanced.

The *irq_desc_t.lock* field

This field provdes a spinlock for the structure, to prevent simultaneous access of the descriptor by multiple microprocessors in an SMP setting.

The hw_irq_controller Structure

Each interrupt descriptor in an irq_desc_t structure contains a field called *`handler`*, a reference to a hw_irq_controller structure. *`Handler`* identifies the code that manages the controller of the interrupt request line. The complete definition of the hw_irq_controller structure is shown in Figure 4.

Figure 4. The hw_irq_controller structure.

```
struct hw_interrupt_type {
  const char * typename;
  unsigned int (*startup)(unsigned int irq);
  void (*shutdown)(unsigned int irq);
  void (*enable)(unsigned int irq);
  void (*disable)(unsigned int irq);
  void (*ack)(unsigned int irq);
  void (*end)(unsigned int irq);
  void (*set_affinity)(unsigned int irq, unsigned long mask);
};

typedef struct hw_interrupt_type hw_irq_controller;
```

The *`hw_irq_controller.typename`* field

The *`typename`* field stores a short, descriptive name for the interrupt controller. This text is used by the kernel's `/proc` directory to display controller-specific interrupt information.

The `hw_irq_controller.startup()` function

Linux calls a controller's `startup()` function when a driver requests an interrupt request line using `request_irq()`, or when interrupts are being probed. `Startup()` is functionally equivalent to `enable()`, except that the former clears any queued or pending interrupts in the controller, while the latter allows pending interrupts to be serviced.

In embedded Linux implementations featuring interrupt controllers that do not queue interrupts, `startup()` and `enable()` are mapped to the same functions in the interrupt controller driver code.

The `hw_irq_controller.shutdown()` function

This function is the complement to the `startup()` function. It is invoked during interrupt probing, and when a driver frees an interrupt request line using `free_irq()`. It is often mapped to the `disable()` function.

The `hw_irq_controller.enable()` function

This function instructs the interrupt controller to enable the requested interrupt line.

The `hw_irq_controller.disable()` function

This function is used to disable the requested interrupt line. It is the functional complement to the `enable()` function.

The `hw_irq_controller.ack()` function

Linux calls an interrupt controller's `ack()` function immediately before servicing the interrupt request. Some controllers don't need this. Some implementations map this function to `disable()`, so that another interrupt request on the line will not cause another interrupt until after the current interrupt request has been serviced.

The `hw_irq_controller.end()` function

When Linux finishes servicing an interrupt request, it invokes the controller's `end()` function. The controller is expected to configure itself as necessary to receive another interrupt request on that line. Some implementations map this function to `enable()`.

The `hw_irq_controller.set_affinity()` function

In SMP environments, this function sets the CPU on which the interrupt will be serviced. This function isn't used in single processor machines.

The `do_IRQ()` Function

The Linux kernel's involvement in interrupt and exception handling begins with the `do_IRQ()` function, which is invoked by the microprocessor's interrupt request handler. The complete source code for the Dreamcast's version of `do_IRQ()` is shown in Figure 5.

There are minor variations in implementations of `do_IRQ()` across platforms and architectures, but in general the code is remarkably consistent because `do_IRQ()` is largely just an irq_desc_t manipulator. The real action takes place in `handle_IRQ_event()`, which is discussed in the next section.

Figure 5. The `do_IRQ()` function.

```
asmlinkage int do_IRQ(unsigned long r4, unsigned long r5,
                      unsigned long r6, unsigned long r7,
                      struct pt_regs regs)
{
  int irq;
  int cpu = smp_processor_id();
  irq_desc_t *desc;
  struct irqaction * action;
  unsigned int status;

  asm volatile("stc  r2_bank, %0\n\t" ❶
    "shlr2  %0\n\t"
    "shlr2  %0\n\t"
    "shlr   %0\n\t"
    "add    #-16, %0\n\t"
      :"=z" (irq));
  irq = irq_demux(irq); ❷

  kstat.irqs[cpu][irq]++; ❸
  desc = irq_desc + irq;
  spin_lock(&desc->lock);

  desc->handler->ack(irq); ❹
  status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
  status |= IRQ_PENDING;

  action = NULL; ❺
  if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS))) {
    action = desc->action;
    status &= ~IRQ_PENDING;
    status |= IRQ_INPROGRESS;
  }
```

```
  desc->status = status;

  if (!action)
    goto out;

  for (;;) { ❻
    spin_unlock(&desc->lock);
    handle_IRQ_event(irq, &regs, action);
    spin_lock(&desc->lock);

    if (!(desc->status & IRQ_PENDING))
      break;
    desc->status &= ~IRQ_PENDING;
  }
  desc->status &= ~IRQ_INPROGRESS;

out:
  desc->handler->end(irq); ❼
  spin_unlock(&desc->lock);

  if (softirq_pending(cpu)) ❽
    do_softirq();
  return 1;
}
```

The Dreamcast version of `do_IRQ()` begins at ❶ by reading part of the identity of the interrupt source into the local variable `irq`, with the help of some inline assembly language. The `irq_demux()` function, ❷, then translates `irq` into an index into `irq_desc[]`, and points the local variable `desc` at the interrupt's associated descriptor. At this point, the interrupt request source has been uniquely identified.

Next, at step ❸, `do_IRQ()` updates the kstat kernel structure to count the interrupt, which updates `/proc/interrupts`. The code then takes a spinlock to protect the interrupt controller hardware from simultaneous access by multiple CPUs.

`Do_IRQ()` acknowledges the interrupt to the requesting source's interrupt controller by calling `desc->handler->ack()` ❹. Some clever code ❺ then sets `action` to either `NULL`, or the head of the list of interrupt handlers registered using the `request_irq()` function. The cleverness is intended to prevent additional microprocessors (if any) from competing to service the interrupt, and to facilitate interrupt line probing by the kernel and device drivers. The interrupt probing process is covered later, in the section called *Interrupt Probing*.

`Do_IRQ()` then enters a loop that invokes `handle_IRQ_event()` multiple times if the interrupt is asserted more than once ❻. The implementation is only of value in an SMP setting, where `handle_IPQ_event()` can be running simultaneously in more than one CPU. In all other cases, `handle_IRQ_event()` will only be invoked once.

Finally, `do_IRQ()` updates the interrupt's *`desc->status`* bitmap to indicate that the interrupt has been serviced, invokes `desc->handler->end()` to signal the interrupt controller that the interrupt has been serviced ❼, and then runs any *softirq*'s associated with the interrupt ❽.

The `handle_IRQ_event()` Function

Once `do_IRQ()` has determined the identity of an interrupt source, the list of interrupt handlers for that source is passed to `handle_IRQ_event()` for processing. The source code for `handle_IRQ_event()` is shown in Figure 6.

Figure 6. The `handle_IRQ_event()` Function

```
int handle_IRQ_event(unsigned int irq, struct pt_regs * regs,
                     struct irqaction * action)
{
  int status;
  int cpu = smp_processor_id();

  irq_enter(cpu, irq); ❶

  status = 1;

  if (!(action->flags & SA_INTERRUPT))
     __sti(); ❷

  do { ❸
    status |= action->flags;
    action->handler(irq, action->dev_id, regs);
    action = action->next;
  } while (action);

  if (status & SA_SAMPLE_RANDOM) ❹
    add_interrupt_randomness(irq);

  __cli();

  irq_exit(cpu, irq);

  return status;
}
```

`Handle_IRQ_event()`'s first step is to invoke the `irq_enter()` macro, which increments a counter, `irqstat.local_irq_count`, to track the number of interrupts being serviced by the current microprocessor ❶. This is obviously only necessary in SMP environments, but for consistency, the Dreamcast code retains this functionality anyway. `Irqstat` and `irq_enter()` are defined in `linux/include/asm-sh/hardirq.h`, with portions in `linux/include/linux/irq_cpustat.h`.

An interrupt handler can request that interrupts be disabled during processing, by setting the `SA_INTERRUPT` bit in the `irqflags` parameter to `request_irq()`. This sets the same bit in *`action->flags`*. If `SA_INTERRUPT` is set, then `handle_IRQ_event()` disables interrupts using the `__sti()` macro ❷. The location of this test implies that if an interrupt source is shared, the first handler to invoke `request_irq()` for that source controls the state of interrupts for all other handlers.

Next, `handle_IRQ_event()` invokes all the interrupt service routines associated with the interrupt, by walking the linked list of handlers provided in the `action` parameter ❸.

Finally, `handle_IRQ_event()` checks to see if the interrupt handler's execution interval is a source of random data. If it is, then the function `add_interrupt_randomness()` is invoked to update the `/dev/random` device ❹. This device is used by cryptographic and other libraries that need truly random data to do their work.

The `interrupt` and `handle_exception` Functions

Now that we've seen the Linux kernel's side of interrupt management, it is time to look at the processor-specific side of the interrupt management process. Actually, we have already seen some processor-specific code in the section called *The hw_irq_controller Structure*, where we covered the API for interrupt controller handler functions. The current section and the next cover the rest of the processor-specific code, namely the code that directly responds to interrupt requests before forwarding those requests on to `do_IRQ()` for handling.

The code for the function `interrupt`, in `arch/sh/kernel/entry.S`, is shown in Figure 7. This code is the Dreamcast microprocessor's main interrupt service routine: the code that the microprocessor invokes when an interrupt request is detected.

Figure 7. The `interrupt` function.

```
interrupt:
    mov.l    2f, k2
    mov.l    3f, k3
    bra      handle_exception
    mov.l    @k2, k2

        .align  2
```

```
2:      .long   INTEVT
3:      .long   ret_from_irq
4:      .long   ret_from_exception
```

As you can see, the code for `interrupt` is simple: it reads the identity of the interrupt source from the microprocessor's `INTEVT` register into register `k2`, loads the address of the function `ret_from_irq` into register `k3`, and branches to `handle_exception`.

The code for `handle_exception`, shown in Figure 8, is a bit more complicated. The additional complication is unrelated to interrupt handling, however. Linux on the Dreamcast implements a *lazy FPU*, where the operating system does not save FPU registers unless it detects that the FPU is in use by more than one process. This improves context switch times, because it avoids moving FPU register values in and out of the coprocessor except when necessary.

Once the lazy FPU code is out of the way (everything prior to ❶), `handle_exception` saves user registers, moves to the kernel's stack space, and then runs the identity of the interrupt source through `exception_handling_table`, to pass control to the appropriate handler ❷. Most of the entries in this table point to `do_IRQ()`; the ones that don't are related to functions that are not modifiable, like low-level processor exception handling, virtual memory management, or floating point coprocessor management. Such functions have their own, dedicated interrupt handling implementations.

A portion of `exception_handling_table` is shown in Figure 9.

Figure 8. The `handle_exception` function.

```
handle_exception:
   ; kernel-to-kernel transition?
   ; if so, there's no need to deal with the FPU
   ; because the kernel doesn't use it
   stc    ssr, k0
   shll   k0
   shll   k0
   bf/s   8f

   ; is the FPU even enabled?
   ; if not, we can skip FPU stuff altogether
   mov    r15, k0
   mov.l  2f, k1
   stc    ssr, k0
   tst    k1, k0
   mov.l  4f, k1
   bf/s   9f

   ; save away FPU stuff
   mov    r15, k0
```

```
   sts.l  fpul, @-r15
   sts.l  fpscr, @-r15
   mov.l  6f, k1
   lds    k1, fpscr
   mov.l  3f, k1
   fmov.s fr15, @-r15
   fmov.s fr14, @-r15
   fmov.s fr13, @-r15
   fmov.s fr12, @-r15
   fmov.s fr11, @-r15
   fmov.s fr10, @-r15
   fmov.s fr9, @-r15
   fmov.s fr8, @-r15
   fmov.s fr7, @-r15
   fmov.s fr6, @-r15
   fmov.s fr5, @-r15
   fmov.s fr4, @-r15
   fmov.s fr3, @-r15
   fmov.s fr2, @-r15
   fmov.s fr1, @-r15
   bra    9f
   fmov.s fr0, @-r15

8: ; we interrupted user code,
   ; move to kernel stack space
   mov    #0x20, k1 ❶
   shll8  k1
   add    current, k1
   mov    k1, r15

   ; move the disable-FPU bitmap into k1
   ; (this will eventually go into SR)
   mov.l  4f, k1

9: ; save user registers
   mov    #-1, k4
   mov.l  k4, @-r15

   sts.l macl, @-r15
   sts.l mach, @-r15
   stc.l gbr, @-r15
   stc.l ssr, @-r15
   sts.l pr, @-r15
   stc.l spc, @-r15
```

```
   ; load up the caller-provided return address
   lds   k3, pr

   mov.l k0, @-r15
   mov.l r14, @-r15
   mov.l r13, @-r15
   mov.l r12, @-r15
   mov.l r11, @-r15
   mov.l r10, @-r15
   mov.l r9, @-r15
   mov.l r8, @-r15

   ; disable interrupts,
   ; change register banks
   stc   sr, r8
   or    k1, r8
   mov.l 5f, k1
   and   k1, r8
   ldc   r8, sr

   ; finish saving registers
   mov.l r7, @-r15
   mov.l r6, @-r15
   mov.l r5, @-r15
   mov.l r4, @-r15
   mov.l r3, @-r15
   mov.l r2, @-r15
   mov.l r1, @-r15
   mov.l r0, @-r15

   ; run the interrupt request
   ; through exception_handling_table
   ; (returns to wherever pr is set to)
   stc   k_ex_code, r8
   shlr2 r8
   shlr  r8
   mov.l 1f, r9
   add   r8, r9
   mov.l @r9, r9
   jmp   @r9 ❷
   nop
        .align  2
1:      .long   SYMBOL_NAME(exception_handling_table)
2:      .long   0x00008000
3:      .long   0x000000f0
```

```
4:      .long   0x000080f0
5:      .long   0xcfffffff
6:      .long   0x00080000
```

Figure 9. The `exception_handling_table` lookup table.

```
.data
ENTRY(exception_handling_table)
  .long  error
  .long  error
  .long  tlb_miss_load
  .long  tlb_miss_store
  .long  initial_page_write
  .long  tlb_protection_violation_load
  .long  tlb_protection_violation_store
  .long  address_error_load
  .long  address_error_store
  .long  do_fpu_error
  .long  error
  .long  system_call
  .long  error
  .long  error
  .long  nmi_slot
  .long  none
  .long  user_break_point_trap
  .long  break_point_trap
  .long  interrupt_table
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
  .long  do_IRQ
...
```

An Interrupt Controller Implementation

The code in Figure 10 manages the Dreamcast's IPR interrupt controller. This code manipulates the IPR peripheral's internal control registers to enable, disable and acknowledge interrupt requests. The end of the listing shows the declaration for the IPR's hw_irq_controller structure.

Figure 10. The Dreamcast IPR interrupt controller code.

```
static unsigned int startup_ipr_irq(unsigned int irq)
{
  enable_ipr_irq(irq);
  return 0;
}

static void disable_ipr_irq(unsigned int irq)
{
  unsigned long val, flags;
  unsigned int addr = ipr_data[irq].addr;
  unsigned short mask = 0xffff ^ (0x0f << ipr_data[irq].shift);

  save_and_cli(flags);
  val = ctrl_inw(addr);
  val &= mask;
  ctrl_outw(val, addr);
  restore_flags(flags);
}

static void enable_ipr_irq(unsigned int irq)
{
  unsigned long val, flags;
  unsigned int addr = ipr_data[irq].addr;
  int priority = ipr_data[irq].priority;
  unsigned short value = (priority << ipr_data[irq].shift);

  /* Set priority in IPR back to original value */
  save_and_cli(flags);
  val = ctrl_inw(addr);
  val |= value;
  ctrl_outw(val, addr);
  restore_flags(flags);
}

static void mask_and_ack_ipr(unsigned int irq)
{
```

```
  disable_ipr_irq(irq);
}

static void end_ipr_irq(unsigned int irq)
{
  if (!(irq_desc[irq].status & (IRQ_DISABLED|IRQ_INPROGRESS)))
    enable_ipr_irq(irq);
}

static hw_irq_controller ipr_irq_type = {
  "IPR-IRQ",
  startup_ipr_irq,
  shutdown_ipr_irq,
  enable_ipr_irq,
  disable_ipr_irq,
  mask_and_ack_ipr,
  end_ipr_irq
};
```

Interrupt Probing

The Linux kernel provides functions for *probing* interrupts: automatically determining which interrupt request line a device is tied to. This capability is provided for device drivers by the kernel's `probe_irq_on()` and `probe_irq_off()` functions.

When probing interrupts, a device driver calls `probe_irq_on()` to tell the kernel that that the probing operation is to begin. The driver subsequently forces the device to generate an interrupt request, then calls `probe_irq_off()` to retrieve the identity of the interrupt request line that the device asserted. An example of how a device driver might use these functions is shown in Figure 11.

Figure 11. An example of how to probe interrupts.

```
void mydriver_probe_irq ( void )
{
  unsigned long mask;
  int irq;
  int retry = 4;

  while( retry-- ) {

    mask = probe_irq_on();
```

```
    /* do something that makes the device
       generate an interrupt request */
    ...

    irq = probe_irq_off( mask );

    if( irq <= 0 )
      printk( KERN_INFO __FUNCTION__
        ": cannot identify interrupt line.\n" );
    else
      printk( KERN_INFO __FUNCTION__
          ": binding handler to interrupt %ld\n", irq );

    if( irq > 0 ) {
      if( request_irq( irq, ... ) <= 0 )
        printk( KERN_INFO __FUNCTION__
          ": could not bind handler to interrupt %ld.\n", irq );
      else break; /* got it! */
    }
  };

  if( !retry || irq <= 0 )
    printk( KERN_INFO __FUNCTION__
      ": could not install interrupt handler, giving up.\n" );

  return;
}
```

The next two sections describe how `probe_irq_on()` and `probe_irq_off()` work. You probably won't need to modify these functions when porting Linux to new hardware, but knowing how they work will help clarify the behavior of `do_IRQ()` and other interrupt-related functions.

The `probe_irq_on()` function

Like `do_IRQ()`, `probe_irq_on()` is largely an irq_desc_t structure manipulator. The code for the Dreamcast version is shown in Figure 12.

Figure 12. The `probe_irq_on()` Function

```
unsigned long probe_irq_on ( void )
{
  unsigned int i;
  unsigned long delay;
```

```
  unsigned long val;

  spin_lock_irq(&irq_controller_lock);
  for (i = NR_IRQS-1; i > 0; i--) { ❶
    if (!irq_desc[i].action) {
      irq_desc[i].status |= IRQ_AUTODETECT | IRQ_WAITING;
      if (irq_desc[i].handler->startup(i)) ❷
        irq_desc[i].status |= IRQ_PENDING;
    }
  }
  spin_unlock_irq(&irq_controller_lock);

  for (delay = jiffies + HZ/10; time_after(delay, jiffies); ) ❸
    synchronize_irq();

  val = 0;
  spin_lock_irq(&irq_controller_lock);
  for (i=0; i<NR_IRQS; i++) {
    unsigned int status = irq_desc[i].status;

    if (!(status & IRQ_AUTODETECT))
      continue;

    if (!(status & IRQ_WAITING)) { ❹
      irq_desc[i].status = status & ~IRQ_AUTODETECT;
      irq_desc[i].handler->shutdown(i);
    }

    if (i < 32) ❺
      val |= 1 << i;
  }
  spin_unlock_irq(&irq_controller_lock);

  return val;
}
```

`Probe_irq_on()` begins by setting the `IRQ_AUTODETECT` and `IRQ_WAITING` flags for each disabled interrupt request line ❶. The `IRQ_DISABLED` flag is already set, because the interrupt line is disabled. Each disabled interrupt request line is then enabled ❷.

Uninitialized or confused hardware devices will often assert spurious interrupt requests, even when there is no handler configured to service the request. `Probe_irq_on()` waits about 100 milliseconds for all such devices to make their interrupt requests ❸, then it clears the `IRQ_AUTODETECT` flag and disables each affected interrupt request line ❹, to take it out of the list of lines that can be probed. Interrupt request lines receiving spurious interrupts cannot be probed, because a device driver can't tell if an

interrupt request is coming from the device it is searching for, or from another, uninitialized device that is using the same line.

To understand in more detail how `probe_irq_on()` detects spurious interrupts, recall that `do_IRQ()` clears an interrupt's `IRQ_WAITING` flag when the interrupt is serviced. `Probe_irq_on()` detects spurious interrupts by watching this flag: if `IRQ_WAITING` suddenly clears after the interrupt is enabled but before the device driver has even had a chance to stimulate the device, the interrupt is deemed to be spurious.

Finally, `probe_irq_on()` builds a bitmap that records each interrupt request line that cannot be probed ❺. This bitmap is used by some versions of the `probe_irq_off()` function.

In a nutshell, at the end of `probe_irq_on()` each interrupt request line that can be probed has its `(IRQ_AUTODETECT|IRQ_DISABLED|IRQ_WAITING)` flags set, and is waiting for the device driver to initiate an interrupt request from the device.

The `probe_irq_off()` Function

When probing for an interrupt request line, a device driver first calls `probe_irq_on()`, to tell Linux that it is preparing to request an interrupt from the device it is searching for. Once the driver requests an interrupt from the device, it calls `probe_irq_off()` to signify the end of the probing operation. `Probe_irq_off()` then returns the identity of the interrupt request line the device is tied to, if it can.

The `probe_irq_on()` call leaves the flags for each interrupt request line's descriptor in the state `(IRQ_AUTODETECT|IRQ_DISABLED|IRQ_WAITING)`, with the knowledge that the lone interrupt request sent from the device will clear the `IRQ_WAITING` flag for the device's associated interrupt request line. `Probe_irq_off()`'s job, therefore, is to simply identify the line with the missing `IRQ_WAITING` flag.

The Dreamcast version of `probe_irq_off()` is shown in Figure 13. There isn't much to discuss: after filtering out the interrupt lines that cannot be probed ❶, `probe_irq_off()` simply finds all the interrupt descriptors without an `IRQ_WAITING` flag ❷. If more than one is discovered, a negative number is returned to signify the error ❸.

Figure 13. The `probe_irq_off()` Function

```
int probe_irq_off ( unsigned long mask )
{
  int i, irq_found, nr_irqs;

  nr_irqs = 0;
  irq_found = 0;
  spin_lock_irq(&irq_controller_lock);
```

```
  for (i=0; i<NR_IRQS; i++) {
    unsigned int status = irq_desc[i].status;

    if (!(status & IRQ_AUTODETECT)) ❶
      continue;

    if (!(status & IRQ_WAITING)) { ❷
      if (!nr_irqs)
        irq_found = i;
      nr_irqs++;
    }

    irq_desc[i].status = status & ~IRQ_AUTODETECT;
    irq_desc[i].handler->shutdown(i);

  }
  spin_unlock_irq(&irq_controller_lock);

  if (nr_irqs > 1) ❸
    irq_found = -irq_found;
  return irq_found;
}
```

What became of the `mask` variable returned from `probe_irq_on()`? The current Dreamcast version of `probe_irq_off()` doesn't need it, because it can identify masked interrupt request lines using just the flags in the interrupt descriptor table. But because `probe_irq_off()` is a kernel API function, the `mask` parameter must remain for consistency with other versions of Linux.

Initialization

Linux initializes interrupt controller harware during the `init_IRQ()` call from `start_kernel()`. The implementation of `init_IRQ()` is always located in the architecture-specific portions of the kernel source tree. `Start_kernel()` is located in `init/main.c`.

Softirqs and Tasklets

Under Linux, a *softirq* is an interrupt handler that runs outside of the normal interrupt context, runs with interrupts enabled, and runs concurrently when necessary. The kernel will run up to one copy of a softirq

on each processor in a system. Softirqs replace the *bottom-half handler* strategy used in Linux 2.2 and older kernels, and are designed to provide more scalable interrupt handling in SMP settings.

A *tasklet* is like a softirq, except that the kernel will not run more than one instance of a tasklet at a time, regardless of the number of processors in a system. The term *tasklet* is somewhat misleading, because tasklets have nothing to do with active processes or tasks in the operating system.

Softirqs and tasklets are implemented using their own data structures and kernel functions, and are largely independent of the kernel's hardware-specific interrupt handling mechanisms. As long as the kernel's `do_IRQ()` implementation calls `do_softirq()`, softirqs and tasklets will work as expected.

Active softirqs and tasklets are managed by the kernel's **ksoftirqd** kernel thread.

Copyright

About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for *Embedded Systems Programming Magazine*, a member of the Advisory Panel for the *Embedded Systems Conference*, maintainer of the Crossgcc FAQ, creator of the *gdbstubs* project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at *http://www.billgatliff.com*.