Interrupts and Interrupt Handlers

Mark McDermott

Outline of This Lecture

- ARM Interrupts
- Interrupts on the iMX21 SoC
- Interrupt handlers
- Writing interrupt handler for iMX21 SoC

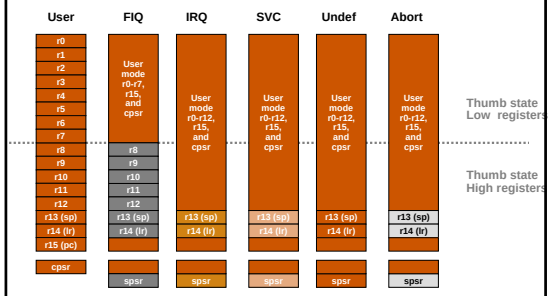
Review of ARM Exceptions

Exception	Description
Reset	Occurs when the processor reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the processor has just powered up. A soft reset can be done by branching to the reset vector (0x0000).
Undefined Instruction	Occurs if neither the processor, or any attached coprocessor, recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction.It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode, such as an RTOS function.
Prefetch Abort	Occurs when the processor attempts to execute an instruction that was not fetched, because the address was illegal[a].
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address[a].
IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Review of ARM Interrupts

- Vector table
  - Reserved area of 32 bytes at the end of the memory map
  - One word of space for each exception type
  - Contains a Branch or Load PC instruction for the exception handler
- Exception modes and registers
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own r13 = stack pointer
    - Its own r14 = link register
    - Its own SPSR (Saved Program Status Register)
  - Exception handlers must save (restore) other register on entry (exit)

Register Organization Summary



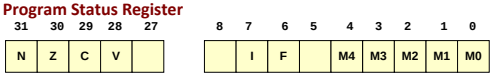
Note: System mode uses the User mode register set

What if Exceptions Happen Simultaneously?

Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

Enabling IRQ and FIQ

- Program Status Register



- To disable interrupts, set corresponding "F" or "I" bit to 1
- On interrupt, processor does the following
  - Switches register banks
  - Copies CPSR to SPSR_mode (saves mode, interrupt flags, etc.)
  - Changes the CPSR mode bits (M[4:0])
  - Disables interrupts
  - Copies PC to R14_mode (to provide return address)
  - Sets the PC to the vector address of the exception handler
- Interrupt handlers must contain code to clear the source of the interrupt

Interrupt Details

- On an IRQ interrupt, the ARM processor will ...
  - If the "I" bit in the CPSR is clear, the current instruction is completed and then the processor will
    - Save the address of the next instruction plus 4 in r14_irq
    - Save the CPSR in the SPSR_irq
    - Force the CPSR mode bits M[4:0] to 10010 (binary)
- This switches the CPU to IRQ mode and then sets the "I" flag to disable further IRQ interrupts
- On an FIQ interrupt, the processor will ...
  - If the "F" bit in the CPSR is clear and the current instruction is completed, the ARM will
    - Save the address of the next instruction plus 4 in r14_fiq
    - Force the CPSR mode bits M[4:0] to 10001 (binary)
      - This switches the CPU to FIQ mode and then sets the "I" and "F" flags to disable further IRQ or FIQ interrupts

IRQ vs. FIQ

- FIQs have higher priority than IRQs
  - When multiple interrupts occur, FIQs get serviced before IRQs
  - Servicing an FIQ causes IRQs to be disabled until the FIQ handler re-enables them
    - CPSR restored from the SPSR at the end of the FIQ handler
- How are FIQs made faster?
  - They have five extra registers at their disposal, allowing them to store status between calls to the handler
  - FIQ vector is the last entry in the vector table
    - The FIQ handler can be placed directly at the vector location and run sequentially after the location
    - Cache-based systems: Vector table + FIQ handler all locked down into one block

Types of Interrupts

Synchronous

Produced by the processor while executing instructions.
Issues only after finishing execution of an instruction.
Often called *exceptions*.
Example: SWI, page faults, system calls, divide by zero

Asynchronous

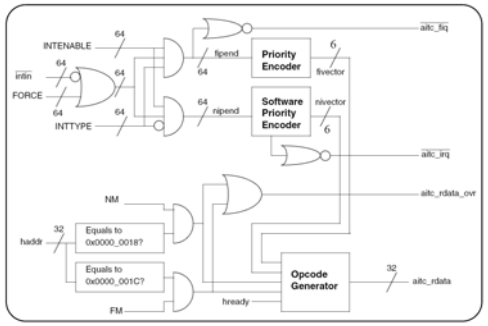
Generated by other hardware devices.
Occur at arbitrary times, including while CPU is busy executing an instruction.
Ex: I/O, timer interrupts

iMX21 Interrupts

iMX21 ARM Interrupt Controller (AITC)

- The AITC performs the following functions:
  - Supports up to 64 interrupt sources
  - Supports fast and normal interrupts
  - Selects normal or fast interrupt request from any interrupt source
  - Indicates pending interrupt sources via a register for normal and fast interrupts
  - Indicates highest priority interrupt number via register (can be used as a table index)
  - Independently enable or disable any interrupt source
  - Provides a mechanism for software to schedule an interrupt
  - Supports up to 16 software controlled priority levels for normal interrupts and priority masking

iMX21 AITC Block Diagram



iMX21 Interrupt Sources (upper 32bits)

Name	Interrupt Source Module	Notes
Reserved Bit 31	Reserved	–
Reserved Bit 30	Reserved	–
INT_LCDC Bit 29	LCD Controller (LCDC)	–
INT_SLCDC Bit 28	Smart LCD Controller (SLCDC)	–
Reserved Bit 27	Reserved	–
INT_USBCTRL Bit 26	–	USB OTG Control Interrupt
INT_USBHP Bit 25	USBOTG	USBOTG HNP Interrupt
INT_USBFUNC Bit 24	USBOTG	USBOTG Function Interrupt
INT_USBHOST Bit 23	USBOTG	USBOTG Host Interrupt
INT_USBDMA Bit 22	USBOTG	USBOTG DMA Interrupt
INT_USBWKUP Bit 21	USBOTG	USBOTG Wakeup Interrupt
INT_EMMAPP Bit 20	eMMA	eMMA Post Processor Interrupt
INT_EMMAPRP Bit 19	eMMA	eMMA Pre Processor Interrupt
INT_EMMADEC Bit 18	eMMA	eMMA Decoder Interrupt
INT_EMMAENC Bit 17	eMMA	eMMA Encoder Interrupt
Reserved Bit 16	Reserved	Reserved for CMMSE
INT_DMACH15 Bit 15	DMA Channel 15	DMA Channel Interrupts
INT_DMACH14 Bit 14	DMA Channel 14	–
INT_DMACH13 Bit 13	DMA Channel 13	–
INT_DMACH12 Bit 12	DMA Channel 12	–
INT_DMACH11 Bit 11	DMA Channel 11	–

Name	Interrupt Source Module	Notes
INT_DMACH10 Bit 10	DMA Channel 10	–
INT_DMACH9 Bit 9	DMA Channel 9	–
INT_DMACH8 Bit 8	DMA Channel 8	–
INT_DMACH7 Bit 7	DMA Channel 7	–
INT_DMACH6 Bit 6	DMA Channel 6	–
INT_DMACH5 Bit 5	DMA Channel 5	–
INT_DMACH4 Bit 4	DMA Channel 4	–
INT_DMACH3 Bit 3	DMA Channel 3	–
INT_DMACH2 Bit 2	DMA Channel 2	–
INT_DMACH1 Bit 1	DMA Channel 1	–
INT_DMACH0 Bit 0	DMA Channel 0	–

iMX21 Interrupt Sources (lower 32bits)

Name	Interrupt Source Module	Notes
INT_CSI Bit 31	CMOS Sensor Interface (CSI)	–
INT_BMI Bit 30	Bus Master Interface (BMI)	–
INT_NFC Bit 29	Nand Flash Controller (NFC)	–
INT_PCMCIA Bit 28	PCMCIA/CF Host Controller (PCMCIA)	–
INT_WDOG Bit 27	Watchdog (WDOG)	–
INT_GPT1 Bit 26	General Purpose Timer (GPT1)	–
INT_GPT2 Bit 25	General Purpose Timer (GPT2)	–
INT_GPT3 Bit 24	General Purpose Timer (GPT3)	–
INT_PWM Bit 23	Pulse Width Modulator (PWM)	–

Name	Interrupt Source Module	Notes
INT_RTC Bit 22	Real Time Clock (RTC)	–
INT_KPP Bit 21	Key Pad Port (KPP)	–
INT_UART1 Bit 20	UART1	–
INT_UART2 Bit 19	UART2	–
INT_UART3 Bit 18	UART3	–
INT_UART4 Bit 17	UART4	–
INT_CSPI1 Bit 16	Configurable SPI (CSPI1)	–
INT_CSPI2 Bit 15	Configurable SPI (CSPI2)	–
INT_SSI1 Bit 14	Synchronous Serial Interface (SSI1)	–
INT_SSI2 Bit 13	Synchronous Serial Interface (SSI2)	–
INT_I2C Bit 12	I2C Bus Controller (I2C)	–
INT_SDHC1 Bit 11	Secure Digital Host Controller (SDHC1)	–
INT_SDHC2 Bit 10	Secure Digital Host Controller (SDHC2)	–
INT_FIR Bit 7	Fast Infra Red Interface (FIR)	–
INT_GPIO Bit 6	General Purpose Input/Output (GPIO)	–
Reserved Bit 5	Reserved	–
INT_CSPI3 Bit 4	Configurable SPI (CSPI3)	–
Reserved Bits 3–0	Unused	–

There are 192 additional interrupts via the GPIO ports.

There are five 32-bit GPIO ports:
Port-A: bits 0-31
Port-B: bits 32-63
Port-C: bits 64-95
Port-D: bits 96-127
Port-E: bits 128-159
Port-F: bits 160-191

Details of the AITC Operation

- The interrupt controller consists of a set of control registers and associated logic to perform interrupt masking, and priority support of normal interrupts.
- The interrupt source registers (INTSRCH / INTSRCL) are a pair of 32-bit status registers with a single interrupt source associated with each of the 64 bits.
- An interrupt line or set of interrupt lines are routed from each interrupt source to the INTSRCH or INTSRCL register. This allows up to 64 distinct interrupt sources in an implementation. Interrupt requests may be forced to be asserted by way of the interrupt force registers (INTFRCH / INTFRCL).
- Each bit in this register is logically “OR-ed” with the corresponding hardware request line prior to feeding the INTSRCH or INTSRCL register inputs.

Details of the AITC Operation (cont)

- There is a corresponding set of interrupt enable registers (INTENABLEH / INTENABLEL), also 32-bits wide which allow individual bit masking of the INTSRCH / INTSRCL registers. There is also a corresponding set of interrupt type register (INTTYPEH / INTTYPEL) which selects whether an interrupt source will generate a normal or fast interrupt to the ARM926EJ-S core.

GPIO Interrupts on the iMX21

- Every general purpose input can be configured as an interrupt and each interrupt can be defined as either:
  - rising-edge triggered
  - falling-edge triggered
  - level sensitive
- The interrupts can be masked using a 32-bit mask register.
- Two levels of interrupt masking are provided. Interrupts can be individually masked at the bit level or at the port level.
- The interrupt status register bits corresponding to the interrupts waiting for service are stored as a value of 1. The interrupt status register is Write 1 to Clear (w1c).

Assigning Interrupt Number on the iMX21

- The 64 interrupt sources are assigned from 0-63 respectively.
- INT-8 (GPIO interrupt) is assigned interrupt numbers 64-255.
- To determine which interrupt sources Linux recognizes type:
  - more /proc/interrupts
- On the TLL-6219 you will see the following 4 interrupt sources:

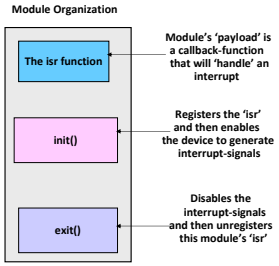
Int #	#INT's	Source
20:	117	IMX-uart
26:	15985	i.MX Timer Tick
55:	0	imx21-hc:usb1
224:	2	smsc911x

- The first 3 sources are internal interrupts. INT-224 is an external interrupt from the Ethernet Controller.
  - The ENET interrupt is connected to Port-F Pin-0 (PF0).
  - This corresponds to bit #160 in the GPIO bit ordering.
  - INT Number = 64 + 160 = 224

Interrupt Handlers

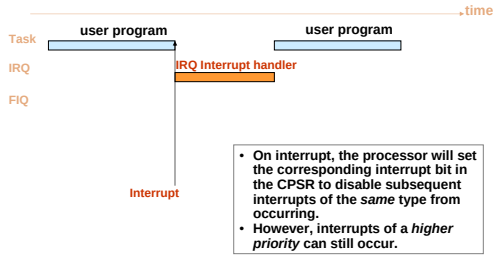
Interrupt Handlers

- Kernel routine that runs in response to interrupt.
  - More than one handler can exist per IRQ.
- Must run quickly.
  - Resume execution of interrupted code.
  - How to deal with high work interrupts?
  - Ex: network, hard disk



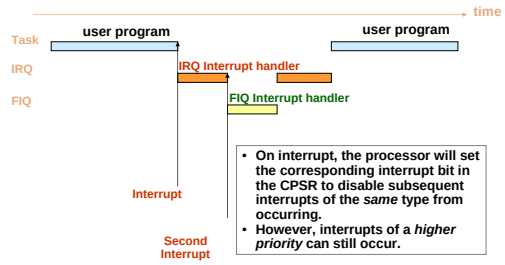
Interrupt Handlers

- When an interrupt occurs, the hardware will jump to an "interrupt handler"



Nested/Re-entrant Interrupts

- Interrupts can occur within interrupt handlers

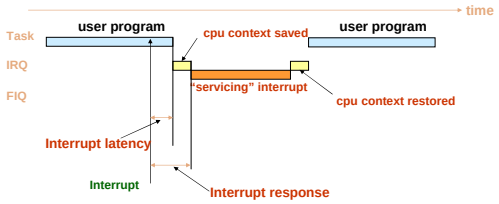


Jumping to the Interrupt Handler

- Auto-vectored
  - Processor-determined address of interrupt handler based on type of interrupt
  - This is what the ARM does
- Vectored
  - Device supplies processor with address of interrupt handler
- Why the different methods?
  - If multiple devices uses the same interrupt type (IRQ vs. FIQ), in an Auto-vectored system the processor must poll each device to determine which device interrupted the processor
    - This can be time-consuming if there is a lot of devices
  - In a vectored system, the processor would just take the address from the device (which dumps the interrupt vector onto a special bus).

Timing of Interrupts

- Before an interrupt handler can do anything, it must save away the current program's registers (if it touches those registers)
- That's why the FIQ has lots of extra registers - to minimize CPU context saving overhead



Interrupt Context

- Not associated with a process.
  - Cannot sleep: no task to reschedule.
  - current macro points to interrupted process.
- Shares kernel stack of interrupted process.
  - Be very frugal in stack usage.

Registering a Handler

request_irq()
- Register an interrupt handler for a given interrupt input pin.

free_irq()
- Unregister a given interrupt handler.
- Disable interrupt line if all handlers unregistered.

Top and Bottom Halves

- Interrupt handling sometimes needs to perform lengthy tasks.
- This problem is resolved by splitting the interrupt handler into two halves:
  - Top half responds to the interrupt
    - The one registered to request_irq
    - Saves data to device-specific buffer and schedules the bottom half
    - Current interrupt disabled, possibly all disabled.
    - Runs in interrupt context, not process context. Can't sleep.
    - Acknowledges receipt of interrupt.
    - Schedules bottom half to run later.
  - Bottom half is scheduled by the top half to execute later
    - With all interrupts enabled
    - Wakes up processes, starts I/O operations, etc.
    - Runs in process context with interrupts enabled.
    - Performs most work required. Can sleep.
    - Ex: copies network data to memory buffers.

Top and Bottom Halves

- Three mechanisms may be used to implement bottom halves
  - SoftIRQs
    - Have strong locking requirements
    - Only used of performance sensitive subsystems – networking, SCSI, etc.
    - Reentrant
  - Tasklets
    - Built on top of SoftIRQs
    - Should not sleep
    - Cannot run in parallel with itself
    - Can run in parallel with other tasklets on SMP systems
    - Guaranteed to run on the same CPU that first scheduled them
  - Workqueues
    - Can sleep
    - Cannot copy data to and from user space

Dos and Don'ts of Interrupt Handlers

- It's a programming offense if your interrupt context code goes to sleep. Interrupt handlers cannot relinquish the processor by calling sleepy functions such as schedule_timeout().
- For protecting critical sections inside interrupt handlers, you can't use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.
- Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch-22, interrupt handlers split their work into two halves: top (slim) and bottom (fat).
- You do NOT need to design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns.
- Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a fast handler.

From: Essential Linux Device Drivers - Venkateswaran

Writing an interrupt handler for the iMX21

Writing an Interrupt Handler for PF16 on iMX21

- The first task to do is to have the driver request the IRQ and associate an interrupt handler with it. This is done as part of init()

```
#define PF16_INT 240     /* PF16 on iMX21 */

static int __init init_interrupt_arm(void)   {
 int rv = 0;

  /* request interrupt  */
 rv = request_irq(PF16_INT, interrupt_interrupt_arm, SA_TRIGGER_RISING | SA_DISABLED,
"interrupt_arm", NULL);

 if ( rv ) {
   printk("Can't get interrupt %d\n", PF16_INT);
   goto no_interrupt_arm;
 }

/* everything is initialized */
 printk(KERN_INFO "%s %s Initialized\n",MODULE_NAME, MODULE_VERSION);
 return 0;

/* free up the irq request on error */
no_interrupt_arm:
    free_irq(PF16_INT,  NULL);
    return -EBUSY;
}
```


Writing an Interrupt Handler for PF16 on iMX21 (cont)

- Interrupt Handler FLAGS
  - The SA_DISABLED flag specifies that this interrupt handler has to be treated as a fast handler, so the kernel has to disable interrupts while invoking the handler.
  - SA_TRIGGER_RISING announces that the pulse input generates a rising edge on the interrupt line when it wants to signal an interrupt. In other words, the pulse input is an edge-sensitive device. Some devices are instead level-sensitive and keep the interrupt line asserted until the CPU services it. To flag an interrupt as level-sensitive, use the SA_TRIGGER_HIGH flag.

From: Essential Linux Device Drivers - Venkateswaran

Writing an Interrupt Handler for PF16 on iMX21 (cont)

- The second task to do is to setup the interrupt handler:

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * function: interrupt_interrupt_arm
 *
 * This function is the interrupt handler for interrupt 240. It flags the
 *  user application that an interrupt occurred.
 */


static struct fasync_struct *fasync_fpga_queue ; // Set up queue to point to calling routine.

void interrupt_interrupt_arm(int irq, void *dev_id, struct pt_regs *regs)

{

/*  Do whatever TOP HALF work needs to be done – quickly */

/* Signal the user application that an interrupt occurred */
    kill_fasync(&fasync_fpga_queue,  SIGIO,  POLL_IN);

return IRQ_HANDLED;   // Exit interrupt handler
}
```

More about kill_fasync() later

Asynchronous Notification

- Polling is inefficient for asynchronous events such as interrupts.
- Solution: Asynchronous notification
  - Application receives a signal whenever data becomes available
  - Two steps
    - Specify a process as the owner of the file (so that the kernel knows whom to notify)
    - Set the FASYNC flag in the device via fcntl() command from the user application system calls:

```
/* create a signal handler */
    signal(SIGIO, &input_handler);
/* set current pid the owner of the stdin */
    fcntl(FILE_DESCRIPTOR, F_SETOWN, getpid());
/* obtain the current file control flags */
    oflags = fcntl(FILE_DESCRIPTOR, F_GETFL);
/* set the asynchronous flag */
    fcntl(FILE_DESCRIPTOR, F_SETFL, oflags | FASYNC);
```


Registering the FILE_DESCRIPTOR

- The character device that is used by user application needs to be registered when the kernel driver is initialized.

```
#define FPGA_MAJOR 245
#define MODULE_NAME "fpga_int"

static int __init init_interrupt_arm(void)   {

if (register_chrdev(FPGA_MAJOR, MODULE_NAME, &fpga_fops))  {
        printk("fpga_int: unable to get major %d. ABORTING!\n", FPGA_MAJOR);
        return -EBUSY;
    }

};
```

- The /dev/fpga_int device is assigned to 245,0
  Use 'mknod /dev/fpga_int c 245 0' to generate the node
- The ***&fpga_ops*** pointer is used to point to the routines that are called when the device is accessed from the user application.

Setting up the file handling operations

- The next step is to assign routines to handle the various device calls through the *fpga_fops* structure.

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*  Define which file operations are supported
*/

struct file_operations   fpga_fops = {
                .owner      =       THIS_MODULE,
                .llseek     =       NULL,
                .read       =       NULL,
                .write      =       NULL,
                .readdir    =       NULL,
                .poll       =       NULL,
                .ioctl      =       NULL,
                .mmap       =       NULL,
                .open       =       fpga_open,
                .flush      =       NULL,
                .release    =       fpga_release,
                .fsync      =       NULL,
                .fasync     =       fpga_fasync,
                .lock       =       NULL,
                .readv      =       NULL,
                .writev     =       NULL,
};
```

This routine handles the device open operations

This routine handles the device close operations

This routine handles all of the async operations for the device

fpga_fasync() routine

This is invoked by the kernel when the user program opens the /dev/fpga_int device and issues fcntl(F_SETFL) on the associated file descriptor.

fasync_helper() ensures that if the driver issues a kill_fasync(), a SIGIO is dispatched to the owning application.

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * function: fpga_fasync
 *
 */

static struct fasync_struct *fasync_fpga_queue ;  // Define queue structure


static int fpga_fasync (int fd, struct file *filp, int on)
{
  /* Register the calling routine in the  fasync_fpga_queue */
     return   fasync_helper(fd, filp, on, &fasync_fpga_queue);
}
```


kill_fasync() routine

- kill_fasync() is used to signal the interested process(es) when data arrives. "kill" is actually a misnomer. This function asynchronously delivers the SIGIO signal to the processes which requested it. Since the default action performed when receiving a signal is to terminate.
- The arguments are the signal to send (usually SIGIO) and the band, which is almost always POLL_IN plus a pointer to the queue with the list of processes to be notified "fasync_fpga_queue"
- Usage:

  kill_fasync(&fasync_fpga_queue, SIGIO, POLL_IN);

Setting up the interrupt from the user space application

- The user space setup involves setting up the actions to be performed and opening the appropriate device.

```
int main(int argc, char **argv)
{
   int count;
   struct sigaction action;   // Setup structure for actions to be performed
   int fd, rc, fc;

   sigemptyset(&action.sa_mask);
   sigaddset(&action.sa_mask, SIGIO);

   action.sa_handler = sighandler;    // Identify the routine to do signal handling
   action.sa_flags = 0;

/* The sigaction system call is used to set the action taken by a process
   on receipt of a specific signal.
*/
   sigaction(SIGIO, &action, NULL);

   fd = open("/dev/fpga_int", O_RDWR);  // Open the device
   fcntl(fd, F_SETOWN, getpid());          // Set the owner of the process
   fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC); // Get and set the flags

/*  User routine follows  ..............  */
```


fcntl() routine

- fcntl((int fd, int cmd) manipulate open file descriptors. It performs one of various miscellaneous operations on fd. The operation in question is determined by cmd:
  - F_GETFL: Read the file descriptor's flags.
  - F_SETFL: Set the file status flags part of the descriptor's flags to the value specified by arg. Remaining bits (access mode, file creation flags) in arg are ignored. On Linux this command can only change the O_APPEND, O_NONBLOCK, O_ASYNC, and O_DIRECT flags.
  - F_SETOWN Set the process ID or process group that will receive SIGIO and SIGURG signals for events on file descriptor fd.

Signal Handling

- The signal handling routine checks to see if the correct signal arrived. A flag can be set to indicate to the main routine that the interrupt happened.

```
int   det_int=0;   // Flag to indicate that interrupt signal detected

void sighandler(int signo)
{
   if (signo==SIGIO) {
      det_int++;

  /*  Perform whatever functions need to be done as part of detecting the interrupt
   *    such as reading data from the device, setting additional flags etc.
   */
}

   return;  /* Return to main loop */
```

main_loop() routine

- This while loop emulates a program running the main loop i.e. sleep(). The main loop is interrupted when the SIGIO signal is received.

```
while(1) {

    /* this only returns if a signal arrives */
    sleep(86400); /* one day */
    if (!det_int)
        continue;

    num_int++;   // Count the number of interrupts – DEBUG ONLY

    printf("mon_interrupt: Number of interrupts detected: %d\n", num_int);

    det_int=0;   // Reset flag for next loop
  }
}
```