Embedded Systems

Input/Output Programming

Outline

- External I/O devices
- I/O software
  - Polled waiting loops
  - Interrupt-driven I/O
  - Direct memory access (DMA)
- Synchronization, transfer rate, and latency
- Interrupt Systems
- Buffers and Queues
- Interrupt Service Routines in C
- NIOS II Interrupt System
External I/O Devices

- External I/O devices in embedded systems are often mechanical in nature
  - Solenoids, relays, etc.
  - Their response time is often characterized by the time required for some physical movement
    - Orders of magnitude slower than that of a typical CPU
- Other I/O devices are analog-to-digital (AD) or digital-to-analog (DA) converters
  - Commonly translate a voltage to/from a digital value
  - Still slower than the typical CPU
- Point to be made:
  - The typical I/O data transfer rate is limited by the speed of the external device, not the CPU
  - A programmer must be certain a CPU does not attempt to transfer data at a rate faster than the device can process it

I/O Device Behavior

- All I/O devices operate asynchronously with respect to the CPU
- Most I/O programming requires some “handshaking” mechanism between the device and the CPU to coordinate reliable data transfer
- The time behavior of I/O data transfer rates vary significantly from one device to another
  - From low data rate, random occurrence (a keyboard for example)
  - To high data rate, periodic occurrence (DMA device)
## Time Behavior in I/O Data Transfers

<table>
<thead>
<tr>
<th>Data Rate</th>
<th>Occurrence</th>
<th>Increasing Time →</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Random</td>
<td>* * * * * * * * * *</td>
</tr>
<tr>
<td></td>
<td>Periodic</td>
<td>* * * * * * * * * *</td>
</tr>
<tr>
<td>High</td>
<td>Random</td>
<td>******** ******** ******** ********</td>
</tr>
<tr>
<td></td>
<td>Periodic</td>
<td>***** ***** ***** ***** ***** *****</td>
</tr>
</tbody>
</table>

---

## Strategies for I/O Software

- I/O software must be designed with careful consideration of requirements and limitations of the I/O device
- Three common strategies
  - Polled waiting loops
  - Interrupt-driven I/O
  - Direct memory access (DMA)
- A selected strategy should be **capable** of data rates fast enough to keep up with the demands of the device, but cannot be allowed to transfer data faster than the device can process it
I/O Device Synchronization

- CPU and I/O devices should be considered physically and logically independent
  - Events that occur within an I/O device that determine when data is available for transfer are usually not under control of the CPU
- Synchronisation is therefore necessary
- Synchronisation steps include:
  - Checking the status of the device
  - Wait until it is ready to transfer data

Transfer Rate

- Transfer rate is a measure of the number of bytes/second transferred between the CPU and an external device
- Maximum transfer rate give a measure of the bandwidth capability of a particular method of performing I/O
  - DMA usually provides the fastest transfer rate, at the expense of additional hardware complexity
  - Polled waiting loops are somewhat slower
  - Interrupt-based I/O can degrade data rates even further unless more than one byte can be transferred per interrupt
    - Due to the overhead of saving/restoring machine state
Latency

- **Latency** is a measure of the delay from the time a device is ready until the first data byte is transferred
  - Essentially equivalent to “response time”
- Latency is:
  - High with polled waiting loops
    - CPU may not be checking the device for data when it is available
  - Average with interrupt-based I/O
    - Still some software overhead
  - Low with DMA

Polled Waiting Loop Pseudocode

```c
IO_input(void) {
    while(((inport(IO_DEVICE_STATUS_PORT) & DEVICE_READY) == 0) {
        // Do nothing (wait for data to arrive)
    }
    return (inport(IO_DEVICE_DATA_PORT));
}

IO_output(unsigned char ch) {
    while(((inport(IO_DEVICE_STATUS_PORT) & DEVICE_READY) == 0) {
        // Do nothing (wait for device to have data)
    }
    outport(IO_DEVICE_DATA_PORT, ch);
}
```

- Memory bandwidth may be a limiting factor for this code
- Spend some time optimizing these types of functions
  - Run from on-chip memory, store results in on-chip memory, etc.
**Hardware Interrupt Processing**

**Hardware interrupt request occurs:** CPU finishes current instruction and initiates an interrupt response sequence

**Interrupt Response Sequence:**
1. CPU pushes return address and status flags to stack
2. Disable interrupts
3. Read interrupt type from requesting device
4. Get ISR address from interrupt vector table
5. Transfer control to ISR

**Interrupt complete:** CPU continues where it left off

**Interrupt Service Routine:**
1. Reenable high priority interrupts
2. Preserve CPU registers
3. Transfer data (clear interrupt)
4. Reenable lower priority interrupts
5. Restore CPU registers
6. Restore status flags and return to interrupted code

---

**Buffers and Queues**

- Interrupt-based I/O inherently separates the ISR that handles the actual I/O transfers from the code that processes the data
- The two codes communicate by passing data through a **buffer**
  - The buffer may be trivial
    - A register or single memory location (variable)
  - A region of memory forming a **queue** (FIFO buffer)
    - A queue helps decouple the timing relationship between the code that fills the queue and that which empties it
    - Allows each piece of code to operate more asynchronously
- Queues may be implemented in software or in hardware
  - AD/DA devices such as an audio coder/decoder may commonly support a hardware queue
Example Queue Operation

- A queue can decouple time of data arrival from time of data processing
- Note the aperiodic arrival rate and the periodic removal rate

| Count | 0 | 1 | 0 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 3 | 2 | 2 | 2 | 1 | 1 | 1 | 0 |
| Insert | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |

C struct for Implementing a Queue

```c
typedef struct {
    unsigned nq;   // insertion index
    unsigned dq;   // removal index
    unsigned count; // #items in queue
    unsigned size; // buffer capacity
    BYTE bfr[1];  // buffer array
} QUEUE;
```

- BYTE is assumed to be an unsigned char
C Code for Creating a Queue

```c
QUEUE *CreateQueue(unsigned bfrsize) {
    unsigned bytes;
    QUEUE *q;
    bytes = (sizeof(QUEUE)-1)+bfrsize;
    q = (QUEUE *) malloc(bytes);
    if (q != NULL) {
        q->nq = q->dq = q->count = 0;
        q->size = bfrsize;
    }
    return q;
}
```

C Code to Queue Data

```c
BOOL Enqueue(QUEUE *q, BYTE data) {
    BOOL full;
    disable(); // Function to disable interrupts
    full = q->count == q->size;
    if (!full) {
        q->bfr[q->nq] = data;
        if (++q->nq == q->size) {
            q->nq = 0;
        }
        q->count++;
    }
    enable(); // Function to enable interrupts
    return !full;
}
```

C Code to Dequeue Data

```c
BOOL Dequeue(QUEUE *q, BYTE *ptr2data) {
    BOOL empty;
    disable();
    empty = q->count == 0;
    if (!empty) {
        *ptr2data = q->bfr[q->dq];
        if (++q->dq == q->size) {
            q->dq = 0;
        }
        q->count--;
    }
    enable();
    return !empty;
}
```

Interrupt Service Routines in C

- Most C compilers provide some support for writing ISRs
- Some compiler support the `interrupt` keyword to identify an ISR

```c
void interrupt ISRfunction(void) {
    // Compiler inserted code to preserve CPU registers
    enable();
    .
    .
    // ISR specific operations, typically to get/send
    // a small amount of data from/to an interrupting device
    .
    .
    // Compiler code to restore saved registers
}
```
NIOS II Interrupt System

- The HAL provides an enhanced application program interface (API) for writing, registering and managing ISRs
  - This API is compatible with both internal and external hardware interrupt controllers.
- Altera also supports a legacy hardware interrupt API
  - This API supports only the internal interrupt controller (IIC)
- Both APIs include the following types of routines:
  - Routines to be called to register an ISR
  - Routines to be called by an ISR to manage its environment
  - Routines to be called to control ISR behavior

The Enhanced HAL Interrupt API

- Enhanced HAL Interrupt API Functions
  - alt_ic_isr_register() – Register a specific ISR function
  - alt_ic_irq_enable() – Enable a specific interrupt
  - alt_ic_irq_disable() – Disable a specific interrupt
  - alt_ic_irq_enabled() – Determine if a specific interrupt is enabled
  - alt_irq_disable_all() – Disable all maskable interrupts
  - alt_irq_enable_all() – Enable all maskable interrupts
Implementing an ISR

• Using the enhanced HAL API to implement ISRs requires the following steps:
  – Write the ISR that handles hardware interrupts for a specific device
    • We will write these in C
  – Register the ISR with the HAL by calling the alt_ic_isr_register() function
    – alt_ic_isr_register() enables hardware interrupts

alt_ic_isr_register()

• The prototype for alt_ic_isr_register() is:
  ```c
  int alt_ic_isr_register(alt_u32 ic_id,
                         alt_u32 irq,
                         alt_isr_func isr,
                         void *isr_context,
                         void* flags)
  ```

• The function has the following parameters:
  – ic_id is the interrupt controller identifier (ID) as defined in system.h
  – irq is the hardware interrupt number for the device
  – isr is a pointer to the ISR function that is called in response to IRQ number irq
  – isr_context points to a data structure associated with the device driver instance. Passed as the input argument to the ISR function to pass context-specific information to the ISR
  – flags is reserved
An ISR to Service a Button PIO Interrupt

- ISR that services a hardware interrupt from a button parallel I/O (PIO) component
  - This example is based on a NIOS II system with a 4-bit PIO peripheral connected to push buttons
- An IRQ is generated any time a button is pushed
- The ISR code reads the PIO peripheral's edge capture register and stores the value to a global variable
  - The address of the global variable is passed to the ISR in the context pointer

```c
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context)
{
    // Cast context to edge_capture's type
    // Volatile to avoid compiler optimization.
    volatile int* edge_capture_ptr = (volatile int*) context;

    // Read the edge capture register on the button PIO and store value
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

    // Write to the edge capture register to reset it
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
}
```
# Code to Register the ISR with the HAL

- The following execution flow is possible:
  - Button is pressed, generating an IRQ
  - The ISR gains control
    - The processor branches to the address in the vector table, which transfers control to the handle_button_interrupts() ISR
  - handle_button_interrupts() services the hardware interrupt and returns
  - Normal program operation continues with an updated value of edge_capture

```c
#include "sys/alt_irq.h"
#include "system.h"
...
// Declare a global variable to hold the edge capture value
volatile int edge_capture;
...
// Initialize the button_pio
static void init_button_pio()
{
    // Recast the edge_capture pointer to match the
    // alt_irq_register() function prototype
    void* edge_capture_ptr = (void*) &edge_capture;
    // Enable all 4 button interrupts
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    // Reset the edge capture register
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    // Register the ISR
    alt_ic_isr_register( BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
                        BUTTON_PIO_IRQ, handle_button_interrupts,
                        edge_capture_ptr, 0x0);
}
```