Computer Labs: Event Driven Design

2º MIEIC

Pedro F. Souto (`pfs@fe.up.pt`)

November 17, 2011

Contents

Events and I/O

Event An **event** is a change in the state

- Virtually all I/O processing is driven by events
  - Whether events are detected by interrupts or by polling
  - Even video graphics output may depend on events (synchronization with the vertical “movement” to avoid visual artifacts)
- Your project will also be event driven:
  - Its execution will depend on events generated by the I/O devices
    - Whether you use polling or interrupts for detecting these events.

Event Driven Design

- This is best addressed by an **event** driven design:
  - That is a design where **flow control** is determined by the **environment** rather than the program itself
  - Essentially, the code is executed **reactively** in response to events that may occur **asynchronously** with program execution
- Event driven design is particularly common in:
  - Graphical user interfaces (GUI)
  - Embedded systems
  - Communications/network software

Simple Event Driven Design

Events The types of events that the different components of the system have to handle

Event Queues That provide the necessary buffering so that handling of an event may occur asynchronously to its occurrence

Event Handlers That process each type of event

Dispatchers That monitor the event queues and call the appropriate event handlers

- May be implemented as a simple loop that checks for events

Event Driven Design and Minix 3 DD Design

- We can find these elements of event driven design in the pattern used in the design of interrupt driven Minix 3 DDs

```
while( 1 ) { /* You may want to use a different condition
    /* Get a request message. */
    if ( driver_receive(ANY, &msg, &ipc_status) != 0 ) {
        printf("driver_receive failed with: %d", r);
        continue;
    }
    if (is_ipc_notify(ipc_status)) { /* received notificat
        switch (_ENDPOINT_P(msg.m_source)) {
        case HARDWARE: /* hardware interrupt notification
            if (msg.NOTIFY_ARG & irq_set) { /* subscribed
                ... /* process it */
            }
            break;
        default:
            break; /* no other notifications expected: do
        }
    } else { /* received a standard message, not a notifica
```

Contents

Event Driven Design and State Machines

- For other than simple designs, it is very helpful to use state machines in combination with event driven design
  - A state machine is useful to handle events that may depend on the state
- A state machine is itself event driven
  - The transition from one state to another depends on the occurrence of an event

Example: Music Playing

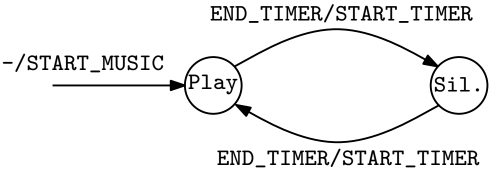
- A music is a succession of notes each with a given duration, with silence intervals in between
- In a game, the music must be played at the same time as there is computer animation on the screen
  - The program cannot stop while a note is being played

Problem How can we measure the time without blocking, as in `tickdelay()`?

Solution A possibility is to use:

- The periodic interruption of the RTC to measure the time, and
- A state machine to keep track of whether we are playing a note or pausing

Example: State Machine



- Should also include:
  - The configuration of Timer 2 with the appropriate frequency
  - The output that enables/disables the speaker
- This state machine is an example of a **Measly Machine**:
  - A state machine where the output depends **not only** on the state **but also** on the input transition
- An alternative state machine is the **Moore Machine**:
  - A state machine where the output depends **only** on the state.
  - This usually leads to extra states
    - In this (simple) case the two machines are structurally equal

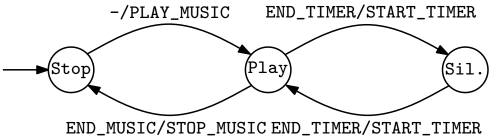
Example: State Transition Table

Cur. State	Input	Next State	Output
Play	END_TIMER	Sil	START_TIMER
Sil	END_TIMER	Play	START_TIMER

- This contains essentially the same information as the state diagram

Example: Implementation Using the RTC's IH

- The IH needs to know whether or not there is a music being played



- The IH needs to keep more state than the state of the state machine itself

  Local state E.g.:
  - The timer value – may be a down counter

  Global sate E.g.
  - The music to be played `music_t`

(Defining a music)

```
/*** Example of a music definition
 *
 * C does not have classes: use structures
 */

typedef struct {
    int freq;    // note's frequency
    int time;    // note's duration
} note_t;

typedef struct {
    int          length; // number of notes in song
    int          cur;    // index of next note to be played
    int          pause;  // pause between notes (in ms)
    note_t       *notes; // pointer to array of notes
} music_t;
```

- Plus a set of methods to simplify the design and the implementation of the system

Example: Playing a Note and “Time” Events

```
music_t *music;                          // the notes queue
enum states {PLAY, PAUSE, STOPPED}; // states of the
                                         //  "playing machine"
void rtc_ih() {
    static int state = STOPPED;      // current state
    static int count; // counts interrupts: keeps track of time
    ...
    if( cause & RTC_PF ) {           // periodic interrupt
        switch(state) {
        case STOPPED:
            if((nt = (note_t *) music_get(music)) != NULL ) {
                timer_load(2, TIMER_CLK/nt->freq);
                speaker_on();
                state = PLAY;
                count = nt->time/RTC_PERIOD;
            }
            break;
        case PLAY:
            count--;
            ...
```

Contents

Event Processing

- I/O devices' events are processed by the corresponding interrupt handlers
- The IHs may be

  Application Dependent
  : As in the case above of the RTC used for playing a music;

  Application Independent
  : Like the code you have developed for the labs ... or may be not

Example

Let's consider a program that plays a note in response to a key pressed in by the user on the keyboard.

IO Devices Keyboard, RTC (and speaker)

Events and event handlers

- Scancodes generated by the keyboard are handled by the KBC IH, and forwarded to the main program
- Time ticks generated by the RTC are handled by the RTC IH

State

- State of the "playing machine"

Example: A Solution

- Keyboard input is handled by an application independent IH
  - Need to define an application dependent event handler
  - Need to specify how the IH “communicates” with this event handler
    - How the data received from the keyboard is passed to the event handler?
    - When is the event handler executed?
- Music playing is handled completely by the RTC IH using a state machine as done above, but with a twist:
  - The notes to play, and the silence duration, are determined by the user rather than by some “music score”.
    - Use a queue for the notes to be played rather than the `struct music`

Example: Handling Events from Keyboard

```
queue_t *kbd_q; // for communication with event handler

setup_keyboard() {
    kbd_q = new_q(...); // setup queue for ih
    set_handler(KBD_IRQ, kbd_ih);
}

void kbd_ih() {
    unsigned long u;
    event |= KBD_EVT; // signal pending event
    sys_inb(RTC_DATA_REG, &u);
    q_put(kbd_q, u); // save in queue
}
```

- `kbd_ih()` is application independent
  - Can be used virtually by all applications
  - Can use the code developed in Lab 4, if ...

Example: Handling Keyboard Events

- Application dependent processing must be performed separately
  - In principle, this code will be different for each application

```
gqueue_t *music_q;  // For passing notes to the RTC IH

void handle_kbd() {
    int n;
    char c;
    q_get(kbd_q, &n);   // not a critical section in Minix 3
    c = scancode2ascii(n);
    switch(c) {
    case 'q':
        exit(0);
    default:
        ...
        gq_put(music_q, note); // put note on a queue
                               // rtc_ih() will play it
    ...
    }
}
```

Minix 3 and Application Independent IHs

```
while( 1 ) { /* You may want to use a different condition
    /* Get a request message. */
    if ( driver_receive(ANY, &msg, &ipc_status) != 0 ) {
        printf("driver_receive failed with: %d", r);
        continue;
    }
    if (is_ipc_notify(ipc_status)) { /* received notificat
        switch (_ENDPOINT_P(msg.m_source)) {
        case HARDWARE: /* hardware interrupt notification
            if (msg.NOTIFY_ARG & irq_set) { /* subscribed
                ... /* process it */
            }
        ...
        }
    } else { /* received a standard message, not a notific
    ...
    }
    /* Now, do application dependent event handling */
    if ( event & KBD_EVT ) {
        handle_kbd();
    } else if ( event &
```

Application Independent vs Application Dependent IH

In General

- Can be reused
  - Operating systems IH is independent of applications
- Introduces a level of indirection
  - May add flexibility
  - May be more responsive
  - Requires more code
  - Has higher overhead

In Minix 3

`driver_receive()` is a blocking call

- Application dependent processing must be done in the same loop iteration as application independent processing
  - It is not possible to delay application dependent processing until there are no interrupts to handle
- It does not afford as much flexibility as in the general case

Thanks to:

I.e. shamelessly translated (some) material by:

- João Cardoso (jcard@fe.up.pt)

Further Reading

- Máquinas de Estado em C