8. Embedded Software

This chapter covers:

- Embedded operating systems
- Device Drivers
- Interrupt handling
- Timers / Watchdogs
- Real-Time features
- Multitasking
- Synchronization

8.1 Embedded Operating Systems

- Embedded Linux / Real-Time Linux / …
- Windows CE
- RoBIOS
- self-made
- none at all!!

Embedded Operating Systems

Important properties of embedded operating sys.

- small (use little memory)
- usually resides in ROM
- provide basic I/O routines for sensors/actuators
- user I/O (if required)
- upgradeability (if required)
- high reliability (for safety critical applications)

Example: RoBIOS

- 128KB
- yes
- routines for motor, servo, PSD, camera, keys, etc.
- LCD and buttons
- yes (download + EEPROM)
- no

Reliability Issue:
Embedded system should never “hang up”

How can this be achieved?

- Simple, well structured, well tested software design
- Avoidance of dynamic data structures
- Software/Hardware watchdogs to test whether processes are still active
- Forced system reset by hardware timer (e.g. every 100ms)
8.2 Device Drivers

A number of I/O routines for a particular device are grouped together as a **device driver**
- In the RoBIOS embedded operating system we use a **Hardware Description Table** for this task
- Each device driver should comprise commands for
  - Initialization of device
  - Decommissioning of device
  - Operation of device
    (e.g. writing to actuator, reading from sensor)
  - Device-specific testing routines

8.3 Interrupts

A data transfer can be
1. CPU-initiated (“polling”)
2. Device-initiated (“interrupt”)

- Execution of one program (user) is **temporarily suspended** for another program with higher priority
  (somewhat like a unscheduled subroutine call)
- Sometimes also called **exception**
- Interrupts can be raised either by **software** (special CPU command) or by **hardware** (external signals linked to CPU interrupt lines)
- Many embedded systems have interrupts that occur at **regular time intervals** (e.g. every 0.01s)
  ⇒ **timer interrupts**
Interrupts

- Often programmed in Assembly (time critical)
- Special return operation required to clear interrupt register (acknowledging that interrupt has been handled): RTE (return from interrupt) instead of RTS (return from subroutine)
- Often CPU registers need to be saved to / restored from stack (time consuming)
- Each interrupt (e.g. interrupt line) has to be associated with the address of an interrupt service routine
  This is done during system setup

Version 1: Handle device without interrupt

Task: Count pulses, e.g. controller: buttons
robot: bumper switch
camera: frame start signal
"You need to be quick ..."

Example Program: Count pulses (active low, use dig. input)
This technique is called “polling”

```
int main()
{
    int counter = 0;
    while(counter < 1000)
    {
        if (!(OSReadInLatch(0) & 0x01))
            counter++;
    }
    return 0;
}
```

Is this correct?

Program is not correct!
because it may count single pulse twice or more!
- Depending on controller speed
- Controller may also miss a short pulse if too slow, but we assume for now this is not the case

We want to count the “0”s
Interrupts

Second Try: Count pulses (active low, use dig. input)

```c
int main()
{
    int counter = 0, previous = 1, now = 1;
    while(counter < 1000)
    {
        now = OSReadInLatch(0)&0x01;
        if (now && previous) counter++; previous = now;
    }
    return 0;
}
```

Is this correct?

Interrupts

Program should work now!

We want to count the "0"s

```c
n=1 n=1 n=1 n=1 n=0 n=0 n=0 n=0 n=0 n=0
p=1 p=1 p=1 p=1 p=0 p=0 p=0 p=0 p=0 p=0
```

if (now == 0 && previous == 1)

Interrupts

Version 2: Handle device with interrupt

Controller has several interrupt lines with different priority

Interrupts

Example Program: Count pulses (use interrupt)

```c
int main()
{
    ???
    return 0;
}
```

How would you do this?
Interrupts

Example Program: Count pulses (use interrupt)
```c
int main()
{
    return 0;
}
```

Unfortunately, this is where C/C++ stops - it does not have language constructs to deal with this.

Therefore ⇒ back to Assembly!

This is where things get complex:
- M68332 has 7 interrupt lines IRQ1’ - IRQ7’
- IRQ1’ is the lowest priority, IRQ7’ the highest
- An IRQ with higher priority can interrupt an IRQ with lower priority
- IRQs 1-6 can be enabled/disabled by software
- IRQ7 cannot be disabled (“nonmaskable interrupt”)
- M68332 supports vectored interrupts and autovectored interrupts

Interrupts

Example Program: Count pulses (use interrupt)

Step 1: Write routine that is called when an interrupt arrives
Step 2: Associate interrupt with this routine (initialization)
Step 3: Enable interrupt

Vectored Interrupts vs. Autovectored Interrupts

**Vectored Interrupt**
- The device supplies an 8 bit value when the CPU acknowledges the interrupt
- This value is used as an index in the system’s interrupt vector table
- Therefore determines which interrupt service routine to call

**Autovectored Interrupt** (simpler)
- No address is supplied
- Use fixed vector address $64 .. $7C
- Activated by
  - Hardware: Set pin $\text{AVEC}$ low during interrupt acknowledge cycle ($\text{or always to Gnd}$)
  - Software: Initialize AVEC Generator in Chip Select Unit
Autovectored Interrupts

Autovectors (IRQ addresses) are located at:

- $0064$ Level 1 Interrupt Autovector  IRQ1'
- $0068$ Level 2 Interrupt Autovector  IRQ2'
- $006C$ Level 3 Interrupt Autovector  IRQ3'
- $0070$ Level 4 Interrupt Autovector  IRQ4'
- $0074$ Level 5 Interrupt Autovector  IRQ5'
- $0078$ Level 6 Interrupt Autovector  IRQ6'
- $007C$ Level 7 Interrupt Autovector  IRQ7'

Interrupts can be triggered either by a low signal on the corresponding IRQ' pin or via software (e.g. TPU interrupts).

Interrupts

Step 1: Write routine that is called when an interrupt arrives
“Interrupt Service Routine”

count_isr:   ADD.L #1, count | increment counter
            RTE      | return interrupt

count:       DC.L 0

Remember to push/pop registers on stack if their contents gets changed!

Step 2: Associate interrupt with this routine (initialization)
As mentioned before:  IRQ1':  $064

MOVE.L  #count_isr, 0x064 | set autointerrupt vector addr.

Step 3: Enable interrupt
Set corresponding bits in status register (set interrupt level to 0)
AND.W   #0xF8FF,SR        |allow all IRQs
## Interrupts

### Status register of 68332

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>T1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>T0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>I2</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>I1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>I0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Trace mode

- **Interrupt Mask**
  - Extend, Negative
  - Zero, Overflow, Carry

#### Supervisor Mode

**AND.W #0xF8FF,SR**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>T1</td>
</tr>
<tr>
<td>14</td>
<td>T0</td>
</tr>
<tr>
<td>13</td>
<td>S</td>
</tr>
<tr>
<td>12</td>
<td>I2</td>
</tr>
<tr>
<td>11</td>
<td>I1</td>
</tr>
<tr>
<td>10</td>
<td>I0</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>Z</td>
</tr>
<tr>
<td>6</td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Only interrupts with a level > interrupt mask level are enabled!**

---

### Timers

**Timer**

```
TimerHandle OSAttachTimer(int scale, TimerFnc function);
```

**Input:**
- (scale) prescale value for 100Hz Timer (1 to ...)
- (TimerFnc) function to be called periodically

**Output:**
- (TimerHandle) handle to reference the IRQ-slot

**Semantics:**
- A value of 0 indicates an error due to a full list (max. 16).
- Attach a irq-routine (void function(void)) to the irq-list.
- The scale parameter adjusts the call frequency (100/scale Hz)
  of this routine to allow many different applications.

---

### Watchdog Timer

**A watchdog is a specialized timer/counter**

- It is initialized to a certain value and keeps counting down
- If the watchdog counter reaches zero, an interrupt is raised
- A correctly running program will reset the watchdog timer in regular intervals to its initial value, so no interrupt will occur
Watchdog Timer

```c
int watchcount;

main()
{
    watchreset();
    OSAttachTimer(1, watchdog);
    ...
    while(1) /* main processing loop */
    {
        watchreset();
    ...
    }
}

void watchdog() /* 100Hz */
{
    watchcount--;
    if (watchcount<0) error("..");
}

void watchreset()
{
    watchcount = 100;
}
```

A watchdog is a very common and very effective tool for fault detection

- Can be used to detect hardware errors and software errors
- Especially useful for if a program “hangs”
- Interrupt or error routine called can reset system or restart individual task

Task Definition

```c
void mytask()
{
    int id, i;
    id = OSGetUID(0); /* id */
    for (i=1; i<=100; i++)
        LCDPrintf("hello%d\n", id);
    OSKill(0); /* terminate */
}
```
Task Start

- Task declaration as `struct tcb`
- Init multitasking as preemptive or cooper.
- Task spawning with name, code, stack size, priority, and id-no.
- Activating task `OSReady`
- Relinquish control `OSReschedule`

Cooperative Multitasking

- Each parallel thread decides itself when to relinquish control
- All threads must be *well behaved*
- `OSReschedule()`

```c
#define SSIZE 4096
struct tcb *task1, *task2;
...
OSMTInit(PREEMPT); /* or coop */
task1 = OSSpawn("t1", task, SSIZE, MIN_PRI, 1);
task2 = OSSpawn("t2", task, SSIZE, MIN_PRI, 2);
if(!task1 || !task2) OSPanic("err");
OSReady(task1); /* set state READY */
OSReady(task2);
OSPermit(); /* activate scheduler */
OSReschedule(); /* goto next task */
...
```

Cooperative Multitasking

```c
#include "eyebot.h"
#define SSIZE 4096
struct tcb *task1, *task2;

void mytask()
{
    int id, i;
    id = OSGetUID(0); /* slave no. */
    for (i=1; i<=100; i++)
    {
        LCDPrintf("task %d : %d\n", id, i);
        OSReschedule(); /* trans. cntrl */
    }
    OSKill(0);  /* terminate thread */
}

int main()
{
    OSMTInit(COOP); /* init multitask. */
    task1 = OSSpawn("t1", mytask, SSIZE, MIN_PRI, 1);
    task2 = OSSpawn("t2", mytask, SSIZE, MIN_PRI, 2);
    if(!task1 || !task2) OSPanic("err");
    OSReady(task1); /* set state READY */
    OSReady(task2);
    OSPermit(); /* activate scheduler */
    OSReschedule(); /* start multitask. */
    /* ----------------------------- */
    /* go HERE when no READY thread */
    LCDPrintf("back to main");
    return 0;
}
```

Cooperative Multitasking

Output cooperative tasks

```
task 2 : 1
task 1 : 1
task 2 : 2
task 1 : 2
task 2 : 3
task 1 : 3
...
task 2 : 100
task 1 : 100
back to main
```
Preemptive Multitasking

- External **scheduler** transfers control between tasks
- Use of time slices
- No need for OSReschedule()

### Code Snippet

```c
void mytask()
{
    int id, i;
    id = OSGetUID(0); /* slave no. */
    for (i=1; i<=100; i++)
    {
        LCDPrintf("task %d : %d\n", id, i);
        OSReschedule(); /* terminate thread */
    }
}
```

Potential problems?

```c
osmtinit (preempt);
...
OSPermit();
```

8.6 Synchronization

**Problem:**
- There is no way of telling when the time slice is up and the next task gets activated
- Task switch might come at a bad time e.g. here in middle of printout

**Solution:**
- **Task synchronization**
  - Semaphores
  - Monitors
  - Message passing
Task Model

Ready

Blocked

Running

start of time slice

or end of time slice

OSReady

or terminate

P(sema)

V(sema)

Semaphores

Semaphores

• Semaphore is data structure for synchronization, with:
  – counter
  – list of blocked processes

• Can be free (1) or blocked (0)

• Binary semaphores vs. counting semaphores

• Block operation P (pass)
  \[ \text{OSSemP} \]

• Free operation V (leave)
  \[ \text{OSSemV} \]

Example Program

```c
struct sem my_sema;
OSSemInit(&my_sema, 1);
...
OSSemP(&my_sema);
/* exclusive block, for example write to screen */
OSSemV(&my_sema);
```

```c
struct tcb *p1, *p2;
struct sem lcd;

int main()
{
  OSMTInit(PREEMPT); /* set multitasking mode */
  OSSemInit(&lcd, 1); /* Init semaphore: allow 1 writer */

  /* Init and ready threads */
  p1 = OSSpawn("P1", myproc, SSIZE, MIN_PRI, 1);
  p2 = OSSpawn("P2", myproc, SSIZE, MIN_PRI, 2);
  OSReady(p1);
  OSReady(p2);

  /* Start multitasking */
  OSPermit(); /* activate multitasking */
  OSReschedule(); /* startup time */

  /* processing will return HERE, when there is no READY thread left */
  return 0;
}
```
Semaphores

Sample Thread - without Semaphore

```c
void myproc()
{
    int id;
    id = OSGetUID(0); /* read thread-no. supplied by main during spawn */
    while(1)
    {
        LCDPrintf("thread %d\n", id);
    }
}
```

Sample Thread - with Semaphore

```c
void myproc()
{
    int id;
    id = OSGetUID(0); /* read thread-no. supplied by main during spawn */
    while(1)
    {
        OSSemP(&lcd);
        LCDPrintf("thread %d\n", id);
        OSSemV(&lcd);
    }
}
```

Reader-Writer Example

/* global */
int stack[1000];
int sindex = -1;

1st try

```c
void writer()
{
    int w;
    for (w=0; w<1000; w++)
    {
        sindex++;
        stack[sindex] = w;
    }
}
```

```c
void reader()
{
    int r;
    for (r=0; r<1000; r++)
    {
        LCDPrintf("%d ", stack[sindex]);
        sindex--;
    }
}
```

2nd try

```c
void writer()
{
    int w;
    for (w=0; w<1000; w++)
    {
        OSSemP(&buf);
        sindex++;
        stack[sindex] = w;
        OSSemV(&buf);
    }
}
```

```c
void reader()
{
    int r;
    for (r=0; r<1000; r++)
    {
        OSSemP(&buf);
        LCDPrintf("%d ", stack[sindex]);
        OSSemV(&buf);
    }
}
```
Reader-Writer Example

/* global */
int stack[1000];
int sindex = -1;
struct sema buf;

void writer()
    { int w;
      for (w=0; w<1000; w++)
      { OSSemP(&buf);
        sindex++;
        stack[sindex] = w;
        OSSemV(&buf);
      }
    }

void reader()
    { int r = 0;
      while (r<1000)
      { OSSemP(&buf);
        if (sindex>=0)
        { LCDPrintf("%d ", stack[sindex]);
          sindex--;
          r++;
        }
        OSSemV(&buf);
      }
    }

Multitasking Operations

int OSMTInit(BYTE mode);
    Input: (mode) operation mode
           Valid values are: COOP=DEFAULT,PREEMPT
    Output: NONE
    Semantics: Initialize multithreading environment

tcb *OSSpawn(char *name,int code,int stksize,int pri,int uid);
    Input: (name) pointer to thread name
           (code) thread start address
           (stksize) size of thread stack
           (pri) thread priority
           (uid) thread user id
    Output: (returncode) pointer to initialized thread
             tcb is initialized and inserted in scheduler queue but not set to READY
    Semantics: Initialize new thread, tcb is initialized and inserted in scheduler queue but not set to READY

int OSReady(struct tcb *thread);
    Input: (thread) pointer to thread control block
    Output: NONE
    Semantics: Set status of given thread to READY

int OSReschedule(void);
    Input: NONE
    Output: NONE
    Semantics: Choose new current thread

int OSGetUID(thread);
    Input: (thread) pointer to thread control block
           (tcb *)0 for current thread
    Output: (returncode) UID of thread
    Semantics: Get the UID of the given thread

int OSKill(struct tcb *thread);
    Input: (thread) pointer to thread control block
    Output: NONE
    Semantics: Remove given thread and reschedule

int OSSleep(int n)
    Input: (n) number of 1/100 secs to sleep
    Output: NONE
    Semantics: Let current thread sleep for at least n*1/100 seconds. In multithreaded mode, this will reschedule another thread. Otherwise, it will call OSWait().
Semaphore Operations

int OSSemInit(struct sem *sem, int val);
Input:   (sem) pointer to a semaphore
(val) start value
Output:  NONE
Semantics: Initialize semaphore with given start value

int OSSemP(struct sem *sem);
Input:   (sem) pointer to a semaphore
Output:  NONE
Semantics: Do semaphore P (down) operation

int OSSemV(struct sem *sem);
Input:   (sem) pointer to a semaphore
Output:  NONE
Semantics: Do semaphore V (up) operation

Concluding Remarks

What are the most important design issues for embedded systems?

- Small, compact
- Cheap
- **Reliable**