Lecture 8: Exceptions, Interrupts, Signals

- Exceptions vs. interrupts
- Programmable interrupt controller
- Interrupt processing sequence
- Interrupt priorities
- VxWorks and interrupt handling
- ISR restrictions
- Signals: concept and operations
- POSIX signal API
- Reentrancy
- Exception handling (Ada/C++/C)
- Using error messages
There is need in a microprocessor system to detect and handle exceptional situations that may change the flow of the program control.

Such situations can be caused by:

- external hardware *interrupts* handled by ISR
- error event *exceptions* (zero-divide, overflow, bus or address error, illegal instruction)
- program event *exceptions* (trap or breakpoint instruction, signal generation)
- Operating system implements selected ISR sending *signals* - the kernel calls `sigInit()` routine at initialization

Interrupts are triggered by external hardware inputs.

Interrupts allow devices to notify the CPU that some event has occurred.

Interrupt processing sequence saves the current context and loads PC with starting address of a user-defined and installed *Interrupt Service Routine*.

- *(ISR)* is kept in the interrupt pointer table in predefined memory locations
- Return from ISR restores saved context to continue execution of the interrupted program sequence
- ISR runs at interrupt time: it is not a task
- On-board timers are a common source of interrupts - using them requires understanding interrupts
Exception Overview

- An exception is an unplanned event generated internally by the CPU (trap or breakpoint instruction, divide by zero, floating point or integer overflow, illegal instruction, or address error)
- VxWorks, as any operating system, installs exception handlers at system startup
- When hardware detects an exception a user-defined exception handler is invoked (if exists)
- A VxWorks exception handler may in turn communicate with a user tasks by signaling semaphore, sending message, generating signal, etc.
- Conceptually, asynchronous interrupts and synchronous exceptions are handled in the same manner

Programmable Interrupt Controller (PIC)

- Interrupts are prioritized based on the classifications
- Most systems have more than one source of interrupt and the sources are prioritized
- PIC is implementation dependent and it serves two main functions:
  - Prioritizing multiple interrupt sources for CPU
  - Offloading the core CPU with the processing required to determine interrupt exact source
- Interrupt Table:
  - Priority - interrupt source priority
  - Vector Address - address of the ISR
  - IRQ - interrupt number
  - Max Frequency - defines time constraint

{Li, Chapter 10, Fig 10.1}
Interrupt Handling Example (Motorola 68k)

Interrupt Vector Table
- vector number
- hardware

Interrupt Service Routine (ISR)
- handler:
- save registers
- call routine
- restore registers
- Return

User ISR
- myISR ( )
  
  {
    ...
  }

Interrupts and their Priorities

Absolute System-Wide Priority

Interrupt Level
(Hardwired)

Execution Order Controlled by Hardware

Execution Order Controlled by Kernel

Task Priority
( Programmable )
Interrupt Stack

- Most architectures use a single dedicated interrupt stack allocated at system start-up
- The interrupt stack size is controlled by the macro `ISR_STACK_SIZE`; default value defined in `configAll.h`

![Diagram of Interrupt Stack]

Hardware Interrupt Handling (1)

- Interrupts are critical in RTS providing links between applications and the controlled hardware
- A system device event can trigger a hardware interrupt and the application must respond to this interrupt within specific determined time constraints
- When an interrupt occurs, it changes the state of the processor causing exception and allowing a user-defined routine to execute in response
- The fastest way to deal with hardware interrupts is running ISR in a special hardware context not involving the overhead associated with a conventional kernel controlled task context switch
- The time between the interrupt generation and the time when the system responds to this interrupt is called **interrupt latency**
Designing an application to handle interrupts requires writing interrupt service routines to handle events.

VxWorks provides routines to support handling hardware interrupts (see sigLib.h):

- **intConnect( )**: connect a C routine to an interrupt vector!!!
  - Takes: VOIDFUNCPTR * vector (interrupt vector to attach to), VOIDFUNCPTR routine (routine to be called, int (parameters for routine)

- **intContext( )**:
  - Returns TRUE only if the current execution state is in interrupt context and not in a task context

- **intLock( ), intUnlock( )**: disable/enable interrupts

- **intVecBaseSet/Get( )**: set/get the vector base address

- **intVecSet/Get( )**: set/get an exception vector

- **ISR code does not run in the normal task context**
  - It has no task control block and all ISR share a single stack

- **ISR must be short, fast, and not invoke functions which may cause blocking of the caller**

**intConnect( ) operation**

Wrapper build by intConnect():

```
myISR
{
    int val;
    ...
    ...
    /* deal with hardware */
    ...
}
```

```
intConnect(INUM_TO_IVEC(someIntNum), myISR, someVal);
```
ISR to Task-Level Code Communication (Allowed???)

- Shared Memory and Ring Buffers:
  - ISR can share variables with task-level code

- Semaphores:
  - ISR can give semaphores (except for mutual-exclusion semaphores) but not take it

- Message Queues:
  - ISR can send messages to message queues for tasks to receive (if the queue is full, the message is discarded) but not receive it

- Pipes:
  - ISR can write messages to pipes that tasks can read; tasks and ISR can write to the same pipes

- Signals:
  - ISR can send "signal" to tasks, causing asynchronous scheduling of their signal handlers

ISR Restrictions

- No tasks can run until ISR has completed

- ISR’s can’t block (it is non-reentrant):
  - Can’t call `semTake()`
  - Can’t call `malloc()` (uses semaphores)
  - Can’t call I/O system routines (e.g. `printf()`)

- The *Programmer’s Guide* gives a list of routines which are callable at interrupt time (some on next slide)

- Typical ISR functions:
  - Reads and writes memory-mapped I/O registers
  - Communicates information to a task by:
    - Writing to memory
    - Making non-blocking writes to a message queue
    - Giving a binary or counting semaphore
Examples of Routines Callable from ISR

- **errnoLib**: errnoGet(), errnoSet()
- **intLib**: intContext(), intCount(), intVecSet(), intVecGet()
- **intArchLib**: intLock(), intUnlock()
- **logLib**: logMsg()
- **msgQLib**: msgQSend()
- **pipeDrv**: write()
- **semLib**: semGive() (except mutex), semFlush()
- **sigLib**: kill()
- **taskLib**: taskSuspend(), taskResume(), taskPrioritySet(), taskPriorityGet(), taskIdVerify(), taskIdDefault(), taskIdReady(), taskIdSuspended(), taskTcb()
- **tickLib**: tickAnnounce(), tickSet(), tickGet()
- **wdLib**: wdStart(), wdCancel()

ISR Guidelines

- Keep ISR short:
  - It does delay lower and equal priority interrupts
  - It does delay all tasks
  - It is hard to debug
- Avoid using floating-point operations in an ISR
  - They may be too slow
  - Must call **fppSave()** and **fppRestore()**
- Try to off-load less critical and/or longer in duration work to application tasks running under kernel control
Debugging ISR

- To log diagnostic information to the console at interrupt time:
  - `logMsg` ("foo = %d\n", foo, 0, 0, 0, 0);
  - Sends a request to `tLogTask` to do a `printf()` for us
- Similar to `printf()`, with the following caveats:
  - Arguments must be four bytes
  - Format string plus six additional arguments
- Use a debugging strategy which provides system-level debugging:
  - WDB agent
  - Emulator

Signals - Concept

- Signals are software representation of interrupts defined inside the operating system and thus they are primary means to notify task about events
- Receiving task must have established a signal handler to catch the signal and respond
  - default exception handler shall handle the exception - in most cases the task shall be suspended and a message is logged on the console
  - Handler is a function that is bound to (or triggered by) the specific signal (handler is an application software version of the Interrupt Service Routine (ISR))
- Signals are, typically, used for error and exception handling (not for a general inter-task communication)
Signals - Basic Terms

• A signal is **generated** when the event that causes the signal occurs.

• A signal is **delivered** when a task or a process takes action based on that signal.

• The **lifetime** of a signal is the interval between its generation and its delivery.

• A signal that has been generated but not yet delivered is **pending** (there may be a considerable time between signal generation and signal delivery).

• A task can block some or all signals with a signal mask, to protect itself from diversions while it executes a critical section of code.

• The exchange of signals provides a means for communication between tasks. (NOTE: message queues, shared memory, and semaphores provide facilities that are generally more suitable for inter-task communication.)

---

Selected Signals (signal.h)

```c
#define SIGINT 2 /* interrupt */
#define SIGQUIT 3 /* quit */
#define SIGILL 4 /* illegal instruction */
#define SIGTRAP 5 /* trace trap */
#define SIGABRT 6 /* used by abort */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination */
#define SIGUSR1 30 /* user defined signal 1 */
#define SIGUSR2 31 /* user defined signal 2 */
#define SIGRTMIN 23 /* Realtime signal min */
#define SIGRTMAX 29 /* Realtime signal max */
```
• An exception in application - when the computer detects an error caused by code that is currently executing (processing cannot continue past the exception point unless the software or a user takes a remedial action)

• In response to the exception, the system software stops the current thread of program execution and generates an appropriate signal

Typical Exception Handling

Task executes

Exception occurs

Is there a signal handler?

Yes

Send the appropriate signal to the task

No

Suspend the task and log a message to the console

Signal handler copes with the situation and returns control to a suitable location in the task

Task continues execution
The Wind kernel supports both BSD 4.3 and standardized POSIX signal interface

Examples of few POSIX1003.1b signal routines from the sigLib library (check signal.h, sigLib.h):

- `signal()`: specify the handler associated with a signal
- `kill()`, `raise()`, `sigqueue()`: send a signal to a task
- `sigaction()`: examine or set a signal handler for a signal
- `sigprocmask()`: examine and/or change the signal mask
- `sigwaitinfo()`: wait for signal

Signals are analogous to hardware interrupts - the basic signal facility provides a set of 31 signals

Signal binding using `signal()`, `sigvec()`, `sigaction()`

Important Caveats

- Signals are **not** recommended for general inter-task communication

- A signal:
  - May be handled at too high a priority if it arrives during a priority inheritance
  - Disrupts a task’s normal execution order. (It is better to create two tasks than to multiplex processing in one task via signals)
  - Can cause reentrancy problems between a task running its signal handler and the same task running its normal code
  - Can be used to tell a task to shut itself down

- sigLib contains both POSIX and BSD UNIX interfaces (do not mix them)
To register a signal handler we may use:

```
signal (signo, handler)
```

- `signo` signal number
- `handler` routine to invoke when signal arrives

returns the installed signal handler (or `SIG_ERR`)

The signal handler is declared as follows:

```
void sigHandler (int sig);  /* signal number */
```

When signal is detected, the offending task will be suspended and a message is logged to the console.

Exception signal handlers typically call:

- `exit()` to terminate the task,
- `taskRestart()` to restart the task, or
- `longjmp()` to resume execution at location saved by `setjmp()`

Hardware exceptions include bus error, address error, divide by zero, floating point overflow, etc.

Some signals correspond to exceptions (e.g. `SIGSEGV` corresponds to a bus error on a 68k).
Example: sending and handling signals (VxWorks)

/* install handlers to intercept signals */
for(sigNo=0; sigNo<32; sigNo++)
signal(sigNo, sigHandler);
...
/* sending all 32 signals to the task specified by taskId*/
sigNo = 1;
while(1)
{
    kill(taskId, sigNo++);
    /* you may use raise(sigNo) */
    if(sigNo>31) break;
    /* limit with 32 signals */
taskDelay(sysClkRateGet()/4);
    /* delay 250 msec */
}

/* handling signals #30 and #31 */
void sigHandler(int sigNo)
{
    logMsg ("%d \n", sigNo, 0, 0, 0, 0, 0);
    switch (sigNo)
    {
    case 30:
        /* respond to signal #30 */
        break;
    case 31:
        /* respond to signal #31 */
        break;
    default:
        printf("\n other signal \n");
    }
}

Reentrancy is critical in code that may be called by more than one task
Example: exception handler (VxWorks)

/* install an exception handler first */
exchHookAdd((FUNCPTR) excHandler);

/* handler code */
void excHandler(void)
{
    logMsg("\n EXCEPTION - RESTART !\n",0,0,0,0,0,0);
    taskRestart(0);
}

Exception Implementation - C++

// the code
try
{
    // any block of code
    catch(<exception_class>)
    { // perform exception handling }
    // function generating exception
    throw(<exception_class>)
}

C++ provides a try block associated with catch and allows for throw calls
Handling exceptions in C is cumbersome and requires preserving the program status by `setjmp` and use `longjmp` to implement exceptions.

### Error Messages

- VxWorks uses an error symbol table (`statSymTbl`) to convert error numbers to error messages.
- To obtain the error string corresponding to `errno`:
  ```c
  { 
    char errStr [NAME_MAX];
    strerror_r (errno, errStr);
    ...
  }
  ```
- To print the error message associated with an error number to the WindSh console:
  ```c
  printErrno(0x110001)
  ```
- `0x110001 = S_memLib_NOT_ENOUGH_MEMORY`
- `value = 0 = 0x0`
VxWorks uses the 32-bit value `errno` as follows:
- Module error numbers are defined in `vwModNum.h`
- Each module defines its own error numbers in its header file
- For example, an `errno` of 0x110001 would be:
  - Module number 0x11 (defined in `vwModNum.h` to be `memLib`) and
  - Error number 0x01 (defined in `memLib.h` to be “not enough memory”)

VxWorks allows to create user-defined error codes

![Module and Error Number Diagram]

**EXAMPLE: Setting errno**

Lowest level routine to detect an error sets `errno` and returns `ERROR`:

```c
STATUS reactorOK() {
    coreTemp = checkCoreTemp();
    if (coreTemp >= maxCoreSafeTemp) {
        errno = S_reactorLib_TEMP_DANGER_ZONE;
        return (ERROR);
    }
    ...
    if (corePressure <= minContainmentPressure) {
        errno = S_reactorLib_LEAK_POSSIBLE;
        return (ERROR);
    }
    ...
}
```
Examine **errno** to find out why a routine failed

```c
if ( reactorOk() == ERROR )
{
    switch (errno)
    {
    case S_rctorLib_TEMP_DANGER_ZONE:
        moveControlRods(0x0f, 0);
        break;
    case S_rctorLib_TEMP_CRITICAL_ZONE:
        logMsg("Run!");
        break;
    case S_rctorLib_LEAK_POSSIBLE:
        checkVessel();
        break;
    default:
        startEmergProc();
    }
    startEmergProc();
}
```

**Summary**

- Exceptions, Signals and Interrupts are means for providing information allowing the program to branch
- Interrupts are handled by hardware
- Signals are RTOS implementation of interrupts
- Interrupt Service Routines have a limited context:
  - No Blocking
  - No I/O system calls
- Using signals for exception handling:
  - `signal( )`
  - `exit( )`
  - `taskRestart( )`
  - `longjmp( )`