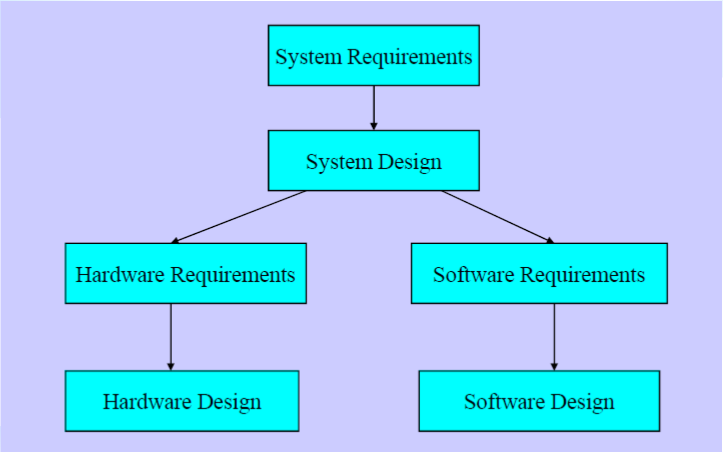
CS 5372
Specification and Design of Real-Time Systems

Lecture 9: Real-Time Software Design Issues

Agenda

- System engineering
- Characteristics of software-intensive systems
- Requirements vs. design
- Modularization: cohesion and coupling
- Development hints: timing, data, events, behavior
- Notations for capturing behavior (in function and in time)
- D/CFD, STD, SD, SC, TD

System Engineering in Product Lifecycle



System Requirements

System Design

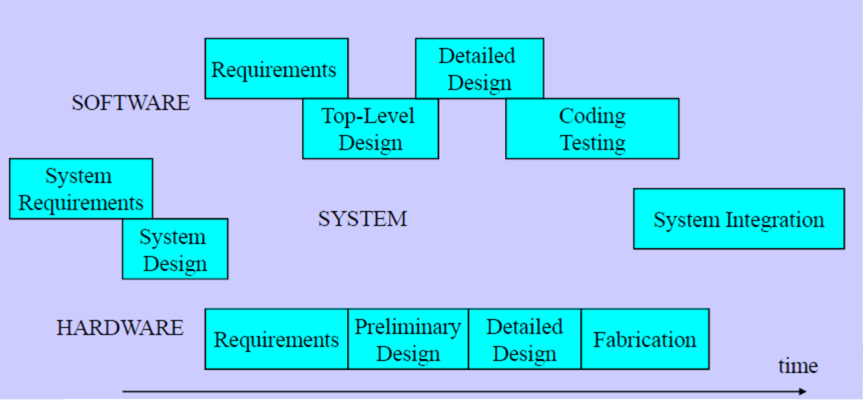
Hardware Requirements

Software Requirements

Hardware Design

Software Design

System Development Process: SW and HW Paths



SOFTWARE

Requirements

Top-Level Design

Detailed Design

Coding Testing

System Requirements

System Design

SYSTEM

System Integration

HARDWARE

Requirements

Preliminary Design

Detailed Design

Fabrication

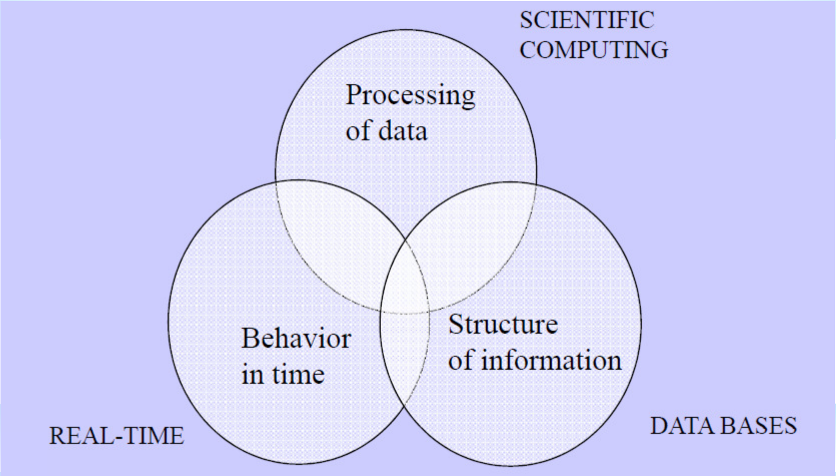
time

Requirements vs. Design

- **Requirements:**
  - Analyze the problem domain and constraints
  - Determine the entities, their functions, interactions, and performance criteria
  - Describe WHAT the software is supposed to do

- **Design:**
  - Define how the software is structured
  - Define the components and their interface
  - Describe HOW to construct the system to meet the requirements

Characteristics of Software Systems



Defining the system

- **System Context:** map of the world interest to the system showing objects external to the system (actors, terminals)
  - capturing the role of environment
  - describing external entities
  - describing events/messages/data exchanged between the system and environment (with their characteristics)

- **Functional approach:**
  - Define function categories (sensing, moving, reporting)
  - Refine each category into sequence of specific functions

How to Identify Requirements?

- Overall system description includes four components:
  - **Function**: what it is supposed to do?
  - **Performance**: how well must it do it?
  - **Constraints**: what can't do?
  - **Interfaces**: how it fits with environment?
    - real-world: operator, physical plant
    - software-world: operating system, databases

- The methods to describe system must be:
  - Formal
  - Visible
  - Expressive
  - Understandable
  - Easy to use

Software Requirements Specification

- Software Requirements Specification's (SRS) objective is to reduce ambiguity of system description
  - translation of the system requirements to software specific requirements
  - the base for design and testing
  - may use varying notations (narrative, diagrams, tables, charts, lists)
  - defines expected behavior, external constraints, and the quality of the system
  - describes inputs, outputs, behavior (in response to events), and performance characteristics (reliability, response time)

Properties of SRS

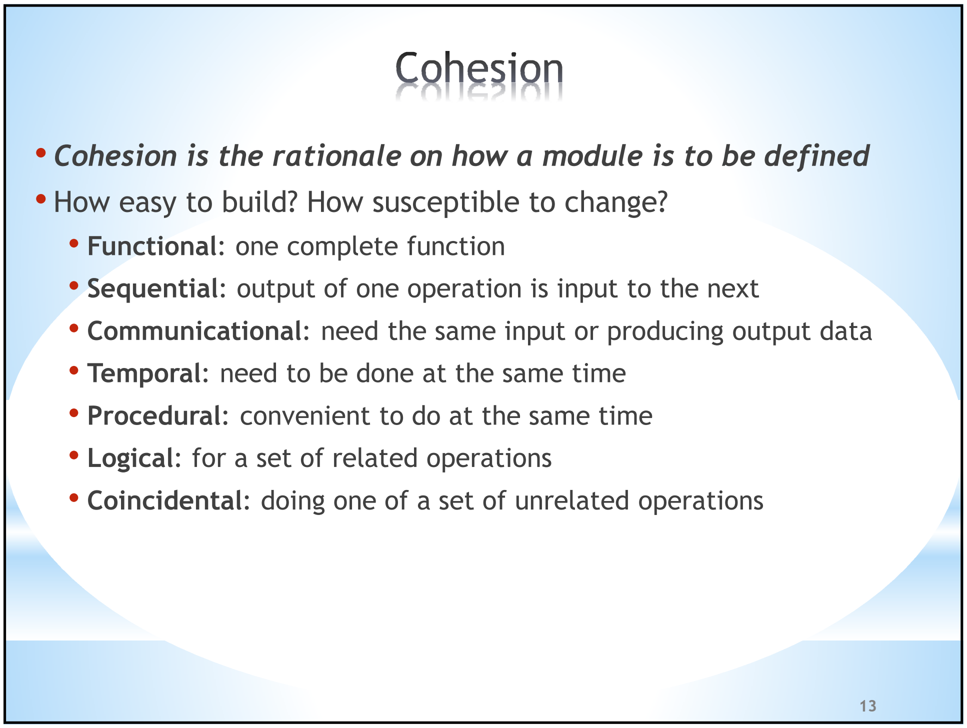
- **Completeness**: inclusion of all the information necessary to develop a specified system
- **Consistency**: no contradictory information in the requirements
- **Correctness**: compliance with external requirements (related documents, standards, scientific knowledge, common sense, etc.)
- **Clarity**: understanding and non-ambiguous interpretation by the external audience
- **Testability**: permits establishment of test criteria and performance of tests to determine whether those criteria have been met
- **Traceability**: relationship between two or more products of the software development process

What is Design ?

- Design is the process of translating the ideas into specific solution consistent with the analysis model
- Design solution must trade-off variety of potential alternatives selecting the best one
- The conventional waterfall model may not be feasible: there is always feedback to earlier lifecycle stages for rework
- Typically, software engineers do NOT design embedded systems, but need to make the system work correctly
- Design methodologies:
  - Functional structuring: abstract machines
  - Object structuring: classes and objects
  - Data flow structuring: data/control flows

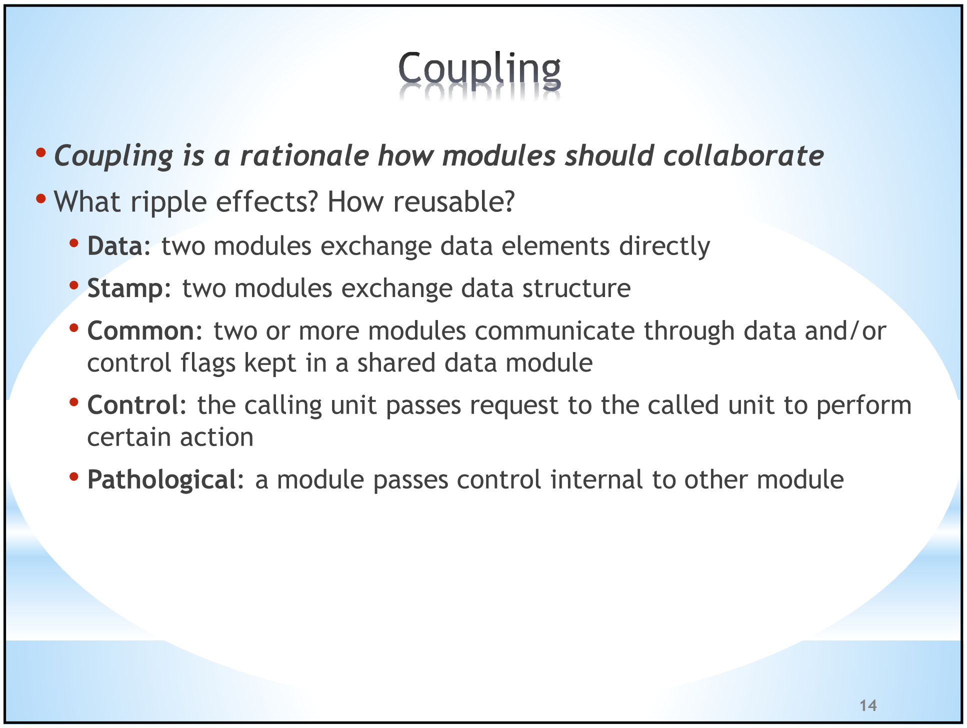
Modularization

- Identification of scope and boundaries of the system and subsystems is critical to limit complexity
- Module selection decisions: partitioning, size, interfaces, complexity, communications, and control
- Modularity support testability, verification, safety
- Module binding is defined by **cohesion**: How the modules are build?
- Module independence is defined by **coupling**: How the modules interface?
- Good design should have a clear description of:
  - Flow of control between modules
  - Partitioning of work between the modules
  - Defining modules interfaces



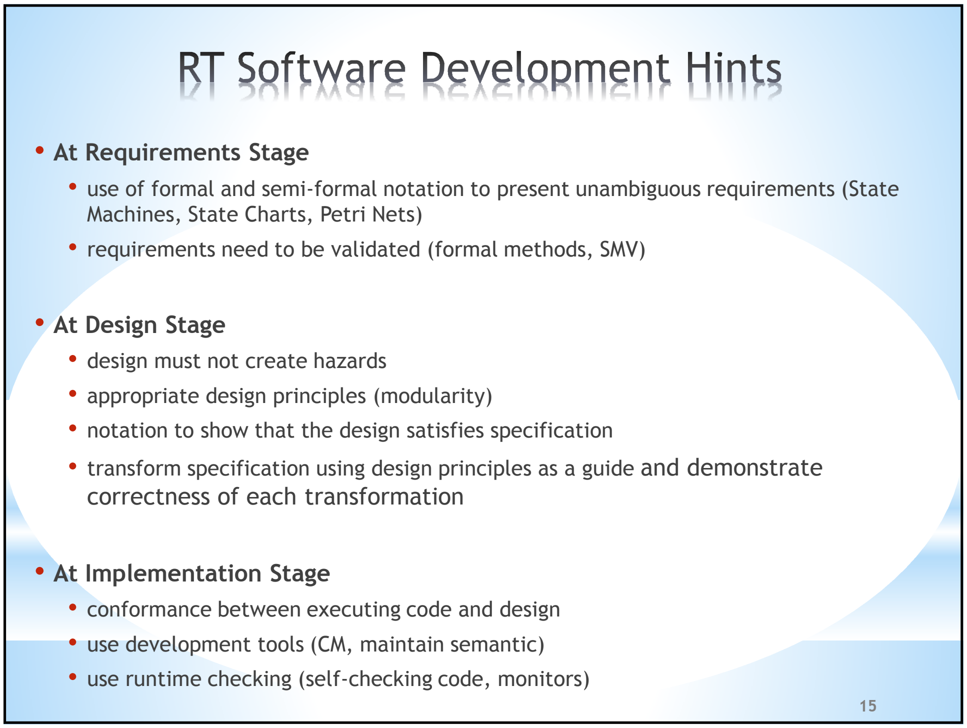
Cohesion

- ***Cohesion is the rationale on how a module is to be defined***
- How easy to build? How susceptible to change?
  - **Functional:** one complete function
  - **Sequential:** output of one operation is input to the next
  - **Communicational:** need the same input or producing output data
  - **Temporal:** need to be done at the same time
  - **Procedural:** convenient to do at the same time
  - **Logical:** for a set of related operations
  - **Coincidental:** doing one of a set of unrelated operations



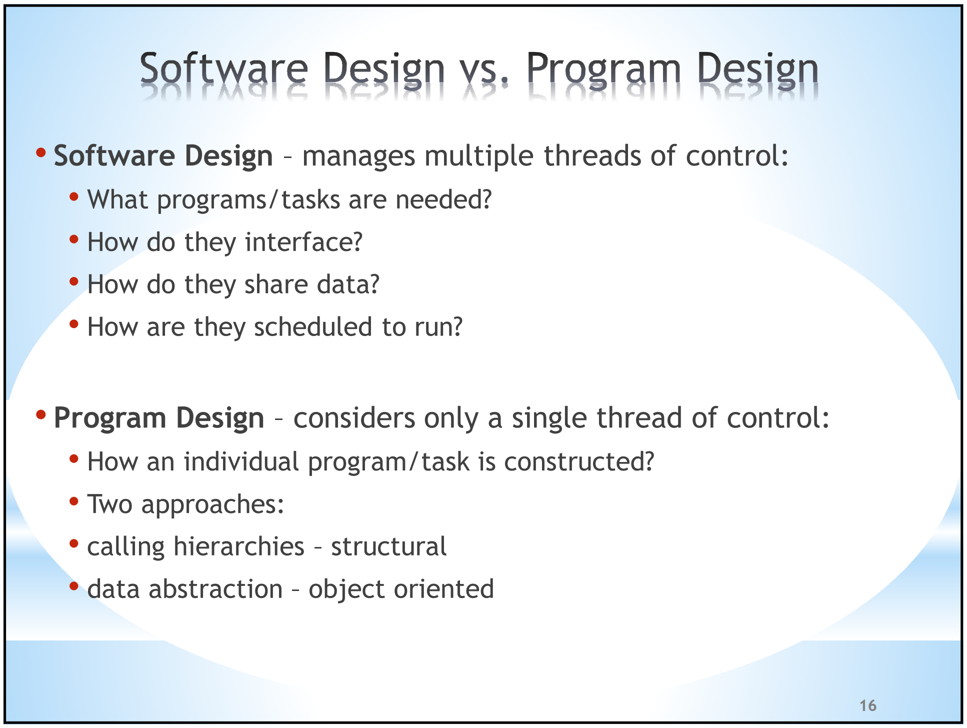
Coupling

- ***Coupling is a rationale how modules should collaborate***
- What ripple effects? How reusable?
  - **Data:** two modules exchange data elements directly
  - **Stamp:** two modules exchange data structure
  - **Common:** two or more modules communicate through data and/or control flags kept in a shared data module
  - **Control:** the calling unit passes request to the called unit to perform certain action
  - **Pathological:** a module passes control internal to other module



RT Software Development Hints

- **At Requirements Stage**
  - use of formal and semi-formal notation to present unambiguous requirements (State Machines, State Charts, Petri Nets)
  - requirements need to be validated (formal methods, SMV)
- **At Design Stage**
  - design must not create hazards
  - appropriate design principles (modularity)
  - notation to show that the design satisfies specification
  - transform specification using design principles as a guide and demonstrate correctness of each transformation
- **At Implementation Stage**
  - conformance between executing code and design
  - use development tools (CM, maintain semantic)
  - use runtime checking (self-checking code, monitors)



Software Design vs. Program Design

- **Software Design** – manages multiple threads of control:
  - What programs/tasks are needed?
  - How do they interface?
  - How do they share data?
  - How are they scheduled to run?
- **Program Design** – considers only a single thread of control:
  - How an individual program/task is constructed?
  - Two approaches:
  - calling hierarchies – structural
  - data abstraction – object oriented

HINT 1: Capture Time and Timeliness

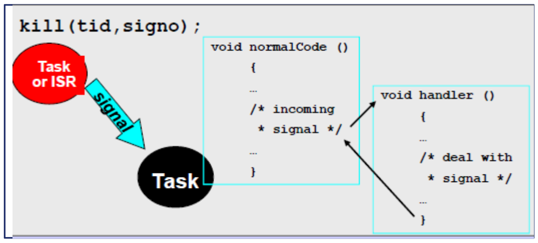
- Most timing requirements are derived arising from need for accuracy or fault tolerance
- Often they are missed leading to unstable system behavior
- Time values must be captured and identified in a form of timing marks (marker bar or relational expression)
- What needs to be identified?
  - incoming periodic messages (period, jitter)
  - incoming sporadic messages (minimum interarrival time, average rate)
  - system response time (deadlines: worst case, average)
  - performance budget (execution times) for the system, including a sub-budget for each activity
  - interrupt and exception handling latencies, context switch time

Selected Signals (signal.h)

```
#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 */
```

Signals - Operations

- 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**

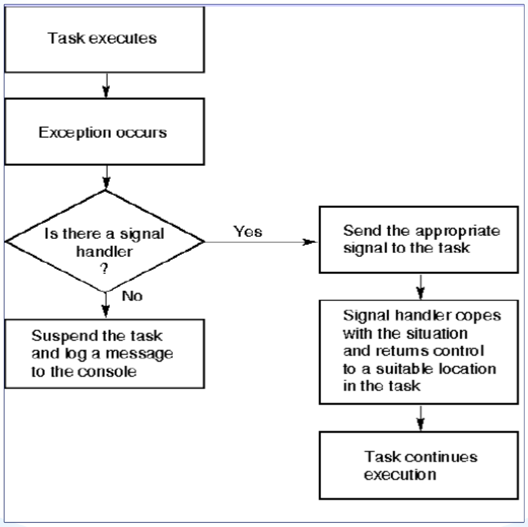


```
kill(tid,signo);
```

```
void normalCode ()
    {
    …
    /* incoming
     * signal */
    …
    }
```

```
void handler ()
    {
    …
    /* deal with
     * signal */
    …
    }
```

Typical Exception Handling



Task executes

Exception occurs

Is there a signal handler ?

Yes

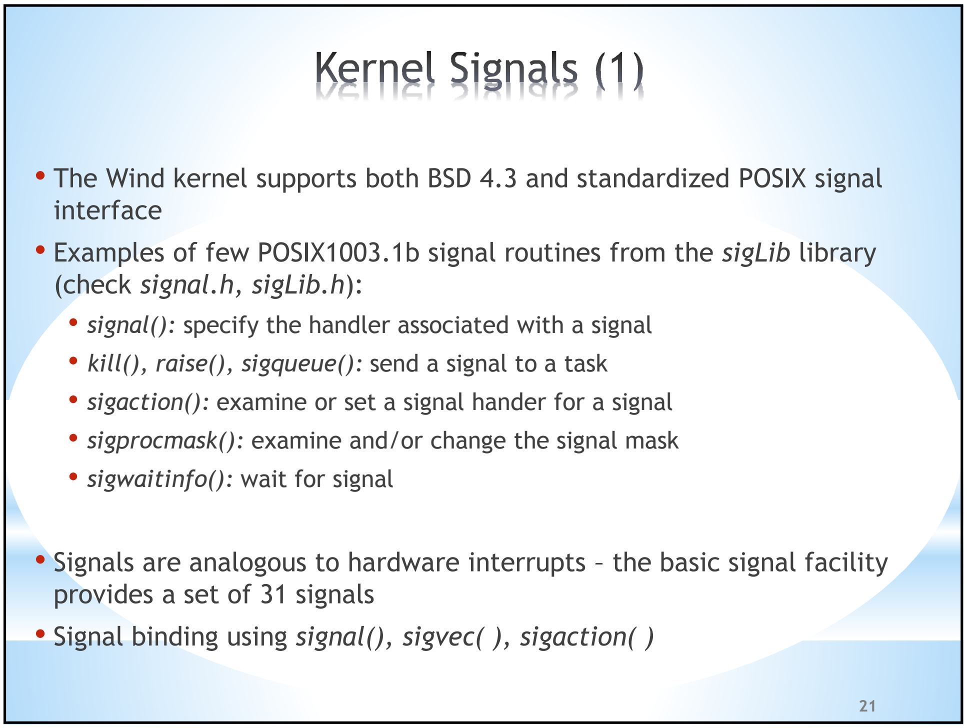
Send the appropriate signal to the task

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

Task continues execution

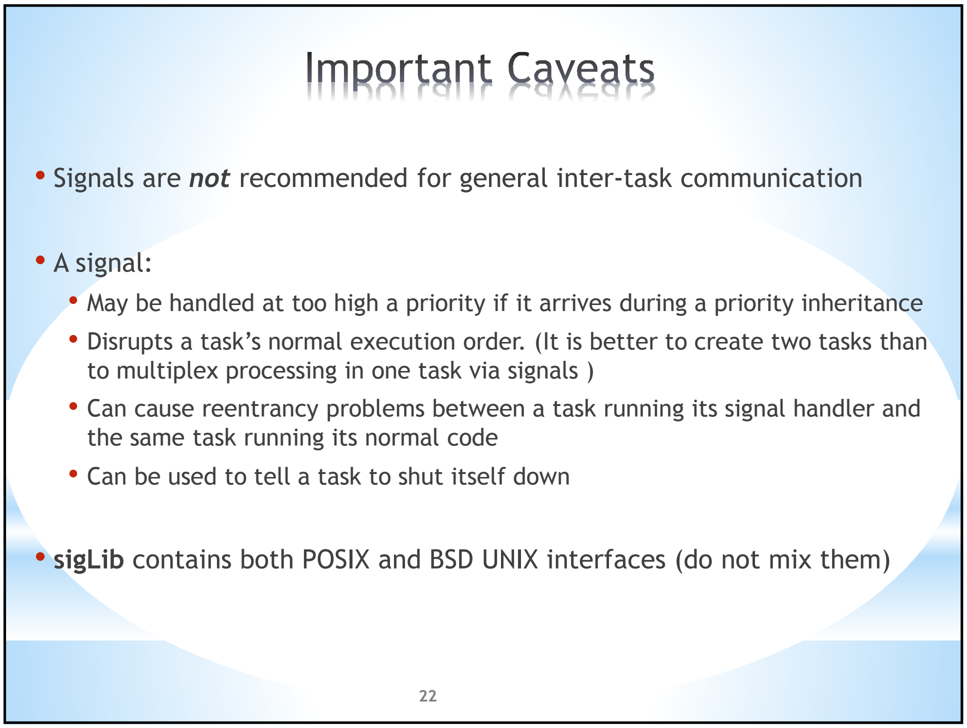
No

Suspend the task and log a message to the console



Kernel Signals (1)

- 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 hander 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)

Signal Handler

- 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()**

Signals and Exceptions

- 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)

myCode( )
{
.
.
.
}

bus error

task has *SIGSEGV* signal handler installed?

YES

NO

raise signal

suspend task
log error message

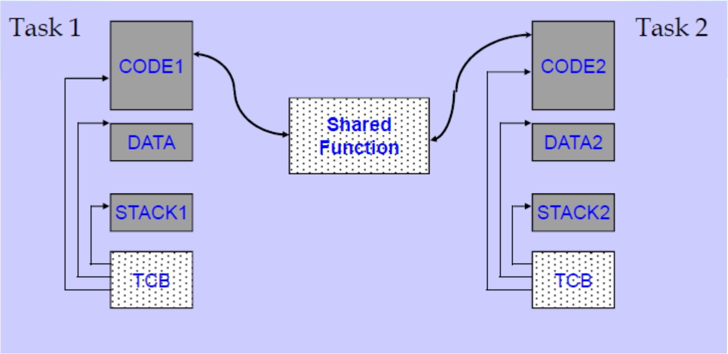
If returns

Example: sending and handling signals (VxWorks)

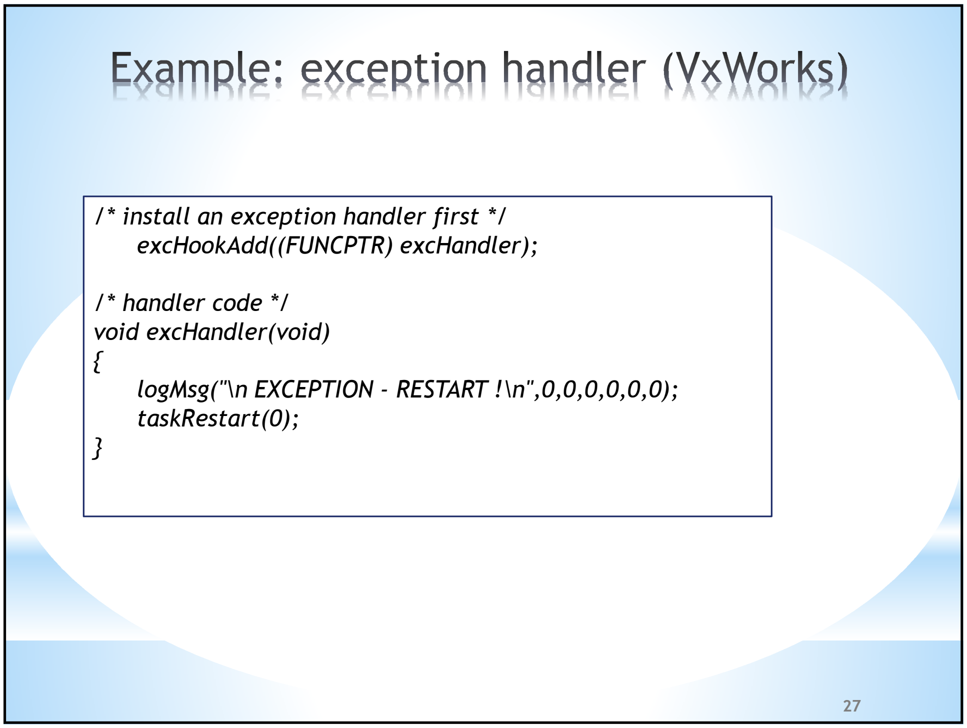
```
/* install handlers to intercept signals */
    for(sigNo=0; sigNo<32; sigNo++)
        signal(i sigHandler);
  …
/* sending all 32 signals to the task
    specified by taskID*/
 sigNo = 1;
 while(1)
    {
       kill(taskID, sigNo++);
           /* you may use raise(sigNo) */
      if(sigNo>31) 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: Sharing Code Between Tasks

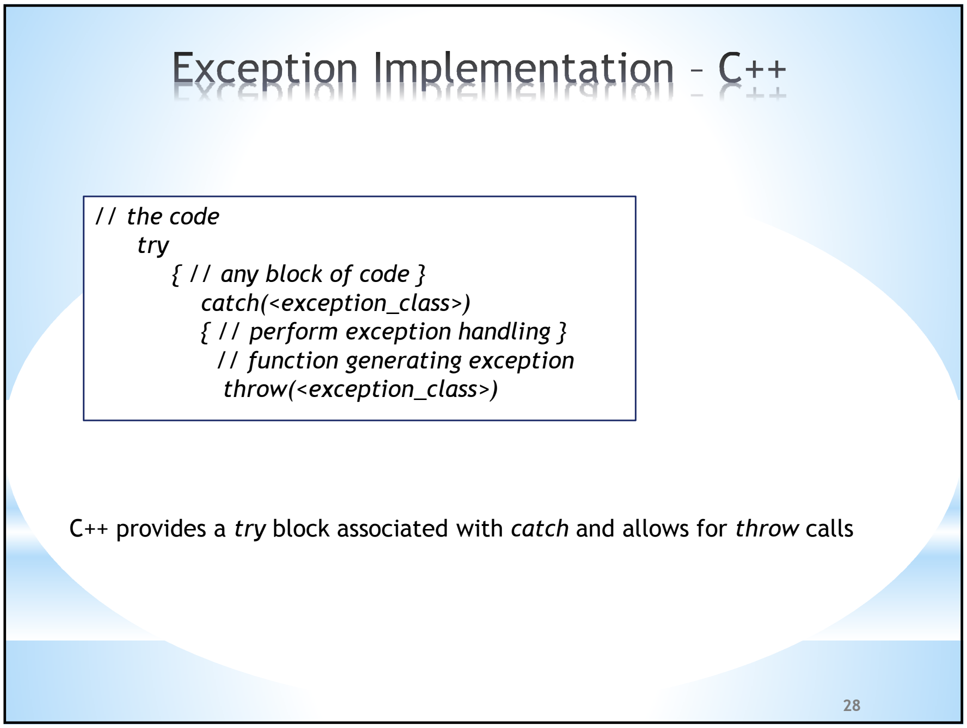


Reentrancy is critical in code that may be called by more than one task



Example: exception handler (VxWorks)

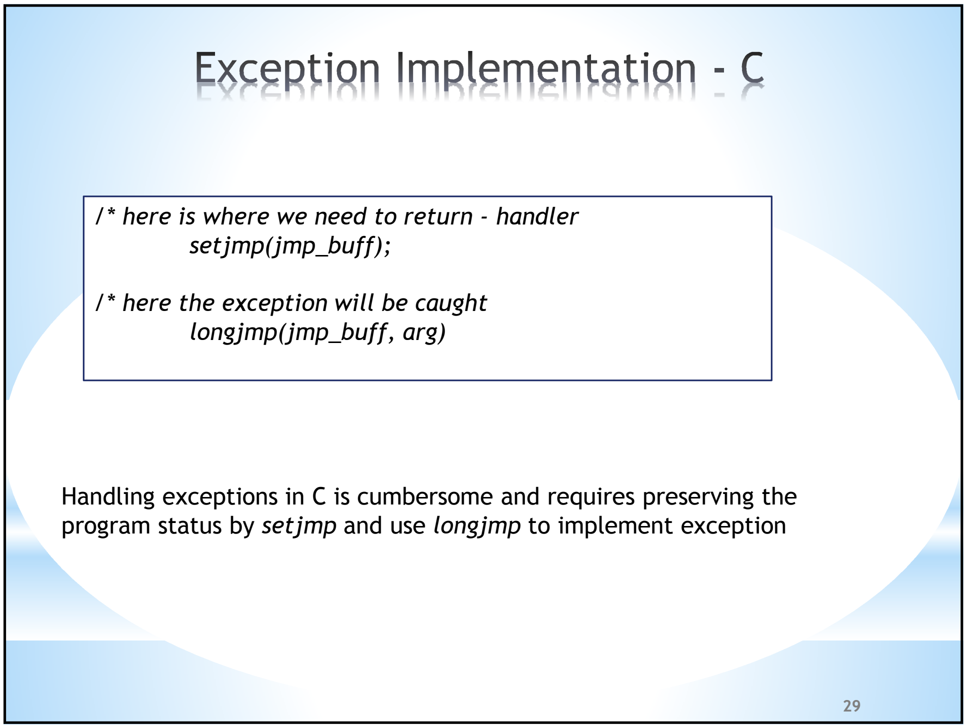
```
/* install an exception handler first */
      excHookAdd((FUNCPTR) excHandler);

/* 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>)
```

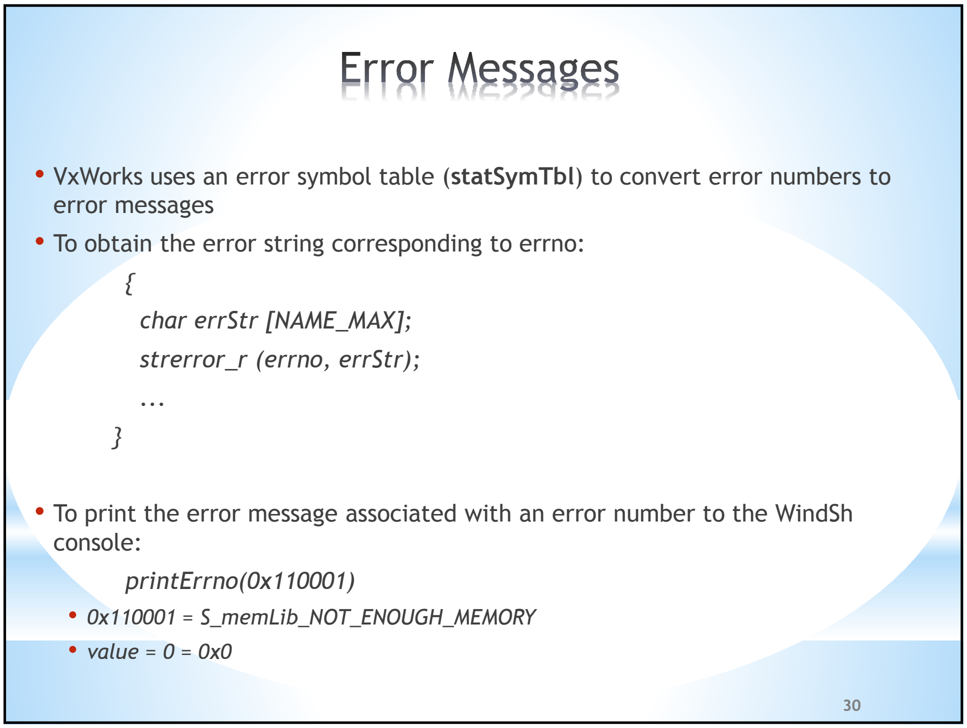
C++ provides a *try* block associated with *catch* and allows for *throw* calls



Exception Implementation - C

```
/* here is where we need to return - handler
        setjmp(jmp_buff);

/* here the exception will be caught
        longjmp(jmp_buff, arg)
```

Handling exceptions in C is cumbersome and requires preserving the program status by *setjmp* and use *longjmp* to implement exception



Error Messages

- VxWorks uses an error symbol table (**statSymTbl**) to convert error numbers to error messages
- To obtain the error string corresponding to errno:

```
    {
      char errStr [NAME_MAX];
      strerror_r (errno, errStr);
      ...
    }
```

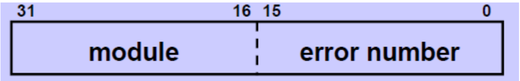
- To print the error message associated with an error number to the WindSh console:

```
    printErrno(0x110001)
```

  - *0x110001 = S_memLib_NOT_ENOUGH_MEMORY*
  - *value = 0 = 0x0*

Interpreting errno

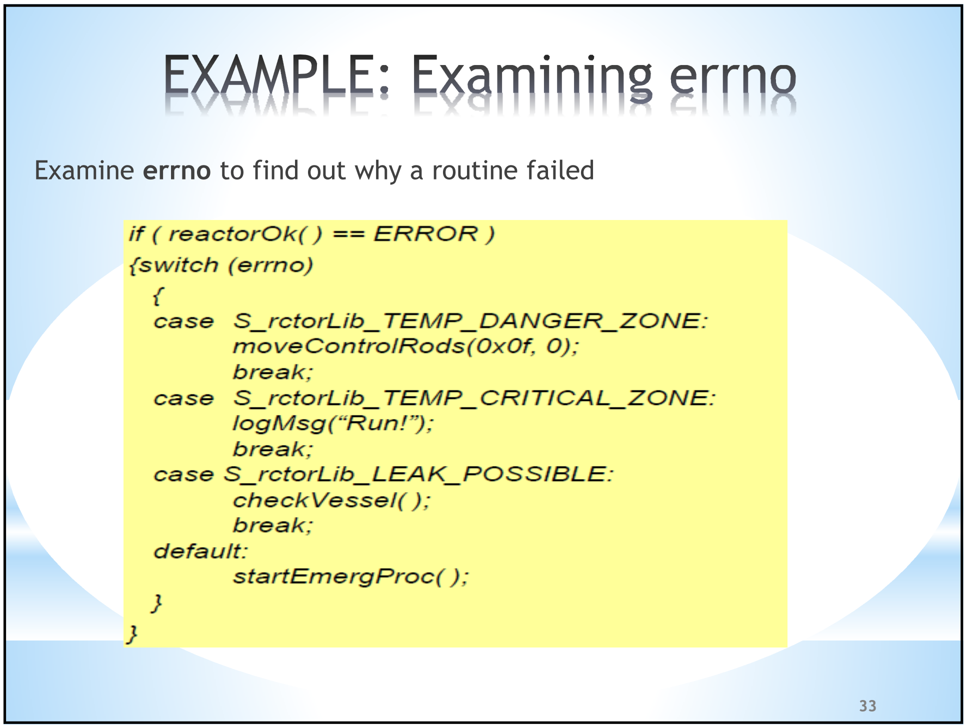
- VxWorks uses the 32-bit value **errno** as follows:
  - **module error numbers** are defined in **vwModNum.h**
  - Each module defines its own error numbers in its header file
  - For example, an **errno** of 0x110001 would be:
    - Module number 0x11 (defined in **vwModNum.h** to be **memLib**) and
    - Error number 0x01 (defined in **memLib.h** to be "not enough memory")
- VxWorks allows to create user-defined error codes



EXAMPLE: Setting errno

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

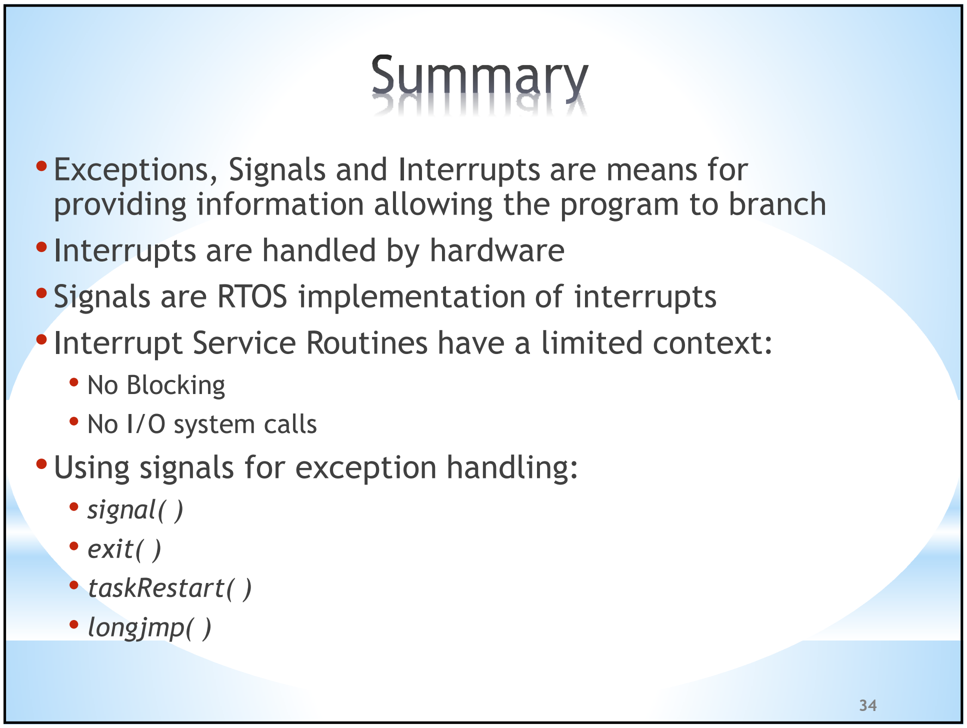
```
STATUS reactorOK( )
  {
  coreTemp = checkCoreTemp( );
  if ( coreTemp >= maxCoreSafeTemp)
         {
                   errno = S_rctorLib_TEMP_DANGER_ZONE;
                   return (ERROR);
         }
         ...
  if ( corePressure <= minContainmentPressure )
         {
                   errno = S_rctorLib_LEAK_POSSIBLE;
                   return (ERROR)
         }
                   ...
  }
```



EXAMPLE: Examining errno

Examine **errno** to find out why a routine failed

```
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( );
  }
}
```



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( )*