# CS 5372
# Specification and Design of Real-Time Systems

Lecture 6: Real-Time Timing + Real-Time Objects

1

## Outline

- Lecture: RT Timing (30)
- Lab Exercise: Watchdog + auxiliary clock timers (25)
- Lecture: RT Objects-Events (15)
- Lab Exercise: Events (30)
- Project Work (rest of session)

2

## Timing

- Time in real-time software deals with:
  - how the software handles time (clocks, delays, timeouts)
  - what are the timing requirements and how to satisfy them (deadlines, execution time, activation rates)
- Real-time software point of interest is real or external time
  - The discrete nature of computer operation requires to consider time granularity and accuracy (to understand **drift** and **jitter**)

3

## Timing/Performance Requirements: Verification

- To capture and record the system timing characteristics we use:
  - **Software** approach – add “time-stamping” code (adds complexity, changes software characteristics, less expensive)
  - **Hardware** approach – add data capturing hardware (requires additional devices, more expensive)
  - **Analytical** approach – model the system in terms of nominal/worst-case analysis (use analytical tools)

4

## How to tell time ?

- Access to time is by concept of a **clock** - usually an external source of interrupts which are counted in a memory location
- A program can access the memory location to find the current time
- A standard UNIX system has time-of-day clock –
  - date +%s returns number of seconds since 01/01/1970
- Other functions must be used to return time in finer granularity (less than one second)

5

## Real-Time and System Clock

- RTC: **Real Time Clock** is integrated with battery powered DRAM to keep track of physical time (CPU independent)
- **System Clock** is a memory location updated on regular basis when system operates
- The system clock initial value is retrieved from the RTC at the power-up
- **Programmable Interval Timer** (PIT) is a device functioning as an event counter (on-chip timer)
  - PIT is controlled via program registers which allows the user to set up the timer interrupt rate to match the desired/available input frequency
- Timer interrupt rate is number of **ticks** (basic unit of time) per second

6

## System Clock

- System clock ISR performs book-keeping:
  - Increments the tick count (use ***tickGet( )*** to examine the count)
  - Updates delays and timeouts
  - Checks for round-robin rescheduling
- These operations may cause a reschedule
- Default clock rate is 60Hz
  - ***sysClkRateSet (freq)*** - sets the clock rate (freq → ticks/second)
  - ***int sysClkRateGet()*** - returns the clock rate
  - ***sysClkRateSet( )*** - should only be called at system start-up
  - tickGet()
  - tickSet(new tick value)

7

## Time Stamp

- **Time stamp** – the actual time in any critical point of the program execution produced by a call to a time access function

```
void timestamp(void)
{
   struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  printf("\nTIMESTAMP:%d sec %d nsec\n", (int)ts.tv_sec, (int)ts.tv_nsec);
}
```

8

# Soft Timer and Clock Resolution

- Soft timer components:
  - timer tick service routine
  - the timer task
- Basic functions: start, stop, update
- RTOS uses timer interrupt to update clock N times a second (which determines clock resolution)
  - in VxWorks: *sysClkRateGet()*

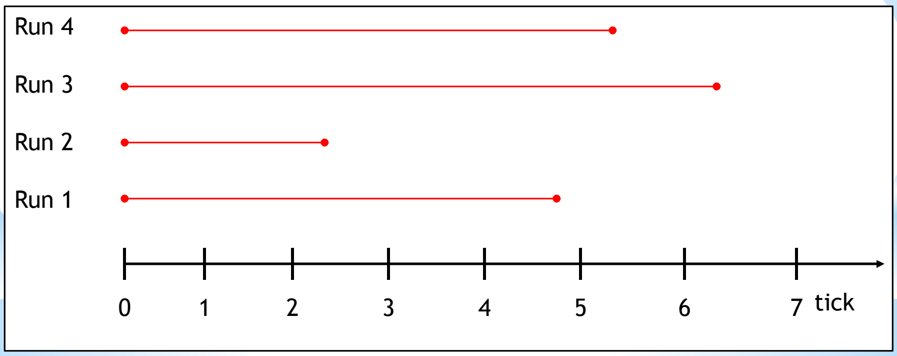


# Example

- What is the resolution in ms of a 100Hz timer interrupt?
  - 100 Hz timer → 100 ticks a second
  - ms = 1/1000 second
- Answer:
  - 10 ms
- Why is this important?
  - any timing request below this resolution will not be met
    - (accuracy is limited to the clock resolution)

## Example: Clock Granularity

- If we have the execution times of four functions runs to be 48, 22, 64, and 52 ms, and if we are using a 100 Hz clock timer, what would you expect the execution time for each to be?




11

## Delays

- We use ***sleep(unsigned int sec)*** to delay program execution by *sec* seconds (it usually is longer; why?)

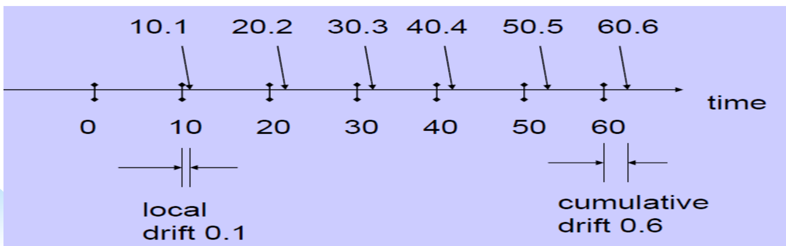
- The "high resolution sleep" POSIX function ***nanosleep( )*** allows sub-second delay with the argument in clock ticks (requires *timerLib*)

- VxWorks specific ***taskDelay(int number)*** *is used* to delay program execution by a *number* of clock ticks (it requires *taskLib)*

12

# Drift

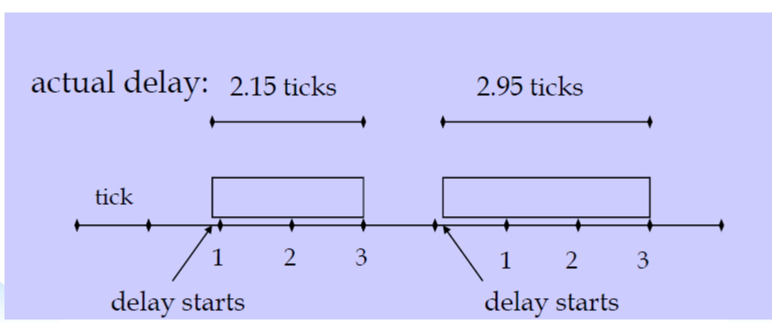
- Delays introduce **local drift** - time overrun associated with delays, resulting in a global **cumulative drift**
- Example:
  - Designed delay interval is exactly 10 ms
  - Actual delay is slightly longer e.g. 10.1 ms




13

# Example: Delay Accuracy

- Delays are accurate to the nearest tick
- *Delay(3)* lasts from the instant it's issued until the third tick arrives




14

## Drift Reduction

- The local drifts may add up to a significant **cumulative drift**
- We compensate for local drift by using modified value of delay interval (reducing the period by a factor related to time of the loop execution)

```
time_base := Clock;
for count in 1 .. MAX_ITERATION loop
   time_old := Clock;  -- remember time before execution
   -- here is the body of code to be repeated every <period>
   -- use either: interval := period - (Clock - time_old);
   --            or: interval := period - (Clock - time_base - (count-1)*period);
   delay (interval);
end loop;
```

15

## VxWorks Timing

- VxWorks provides IEEE standard POSIX 1003.1b clock:
- one *clock_id* is supported,
  - *CLOCK_REALTIME* (additional "virtual" clocks or auxiliary clock hardware could be supported)
  - defined in *time.h*
  - it is a system-wide real-time clock used as an argument in timing function calls
  - VxWorks provides routines to access the clock
  - Example:
    - *clock_gettime(CLOCK_REALTIME, &timeptr)*
    - NOTE: where *timeptr* is of the type *timespec*

16

## VxWorks Timing *Utility*

- *timexLib.h* - contains prototype routines for timing the execution of programs
  - individual functions, and groups of functions (VxWorks system clock is used as a time base)
- These routines can be used also from shell command line:
  - ***timex*( ):** to time a single execution of a function or a group of functions
  - ***timexN*( ):** to time repeated executions of a function or group of functions (N-times)
  - Why are these important?

17

## VxWorks Timing Utility (cont)

- VxWorks *timexLib* additional functions:
  - ***timexInit( )*** - include the execution timer library
  - ***timexClear( )*** - clear the list of function calls to be timed
  - ***timexFunc( )*** - specify functions to be timed
  - ***timexHelp( )*** - display execution timer facilities
  - ***timexPost( )*** - specify functions to be called after timing
  - ***timexPre( )*** - specify functions to be called prior to timing
  - ***timexShow( )*** - display the list of function calls to be timed
- Up to four functions can be specified to be timed as a group
- Up to four functions can be specified as pre- or post-timing functions, to be executed before and after the timed functions

18

## VxWorks tickLib

- VxWorks *tickLib* provides clock tick support
  - ***tickAnnounce( )*** - announce a clock tick to the kernel
  - ***tickSet( )*** - set the value of the kernel's tick counter
  - ***tickGet( )*** - get the value of the kernel's tick counter
- *tickLib* is used mostly as interface to the VxWorks kernel routines that rely on clock ticks, for example: *taskDelay( ), kernelTimeslice( ), wdStart( )*
- It is not appropriate for lengthy time-outs or time-keeping

19

## Timeouts and Timers

- **Timeout is** restriction of time the process can spend in particular code segment, e.g. waiting for something
- In C we use **timers** (or **watchdogs**) to notify the process when certain time expires
- On-board timers interrupt the CPU periodically
- **Timer** is a software that allows user-defined routines to be executed at periodic intervals, useful for:
  - Polling hardware
  - Checking for system errors
  - Aborting an untimely operation
- VxWorks supplies a generic interface to manipulate two timers:
  - System clock
  - Auxiliary clock (if available)

20

## Watchdog Timers

- **Watchdog Timer** (defined in *wdLib*) is a mechanism allowing any C function to be executed after a specified time delay
- Functions invoked by watchdog timers execute as Interrupt Service Routine at the interrupt level (more about it later)
- *wdShow* library contains useful functions to show watchdog statistics, activities, etc.
- Watchdog argument is in clock "**ticks**"

21

## Watchdog Timers Operations

- To create a watchdog timer (returns ID or ***NULL***):
  - ***WDOG_ID wdCreate ()***
- To de-allocate a watchdog timer (and cancel any previous start):
  - ***STATUS wdDelete (wdId)***
- To start (or restart) a watchdog timer:
  - ***STATUS wdStart (wdId, delay, pRoutine, parameter)***

    where
    - ***wdId*** Watchdog id, returned from wdCreate( )
    - ***delay*** Number of ticks to delay
    - ***pRoutine*** Routine to call when delay has expired
    - ***parameter*** Argument to pass to routine
- To cancel a previously started watchdog:
  - ***STATUS wdCancel (wdId)***

22

# Using Watchdogs

Create a WD

When to start the WD

```
wdId = wdCreate();
wdStart (wdId, DELAY_PERIOD, myWdIsr, 0);

void myWdIsr(int param)
	{
	work(param);
	wdStart (wdId, DELAY_PERIOD,myWdIsr, param);
	}
```

Which routine and param. to start to start the WD

Which WD to start

- What does this do?
- The *work( )* routine might:
  - Poll some hardware device
  - Unblock some task
  - Check if system errors are present

23

# Using Watchdogs: What does this one do?

```
WDOG_ID wdId;

void foo(void)
	{
	wdId = wdCreate( );
	/* Must finish each cycle in under 10 seconds */
	while(1)
		{
		wdStart (wdId, DELAY_10_SEC, fooMisDeadHandle, 0);
		fooDoWork( );
		}
	}

void fooMisDeadHandle (int param)
	{
	/* Handle missed deadline */
	...
	}
```

Recover From a Missed Deadline

24

## Summary

- Timing is the critical element of real-time programming
- The programmers need to know how to tell time and how to produce delays
- Modern processors have hardware component providing time information (clock, real-time clock)
- Clock resolution and accuracy must be known
- Drift may have significant effect on program timing
- Watchdogs are a major mechanisms to provide time-related services

25

## Watchdog Timer in-class lab

26

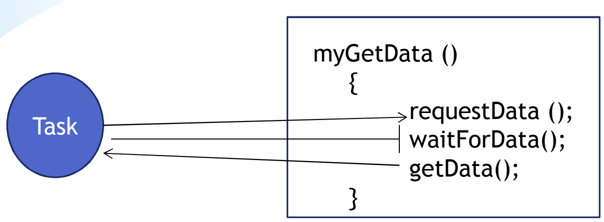
# Real-Time Objects

- Pipes (previous lecture)
- Event Registers



# Event Registers

- A Synchronization Problem:




- Task may need to wait for an event to occur (data becomes available in the example above.
- Busy waiting (polling) is a waste of resources
- Pending until the event occurs is better

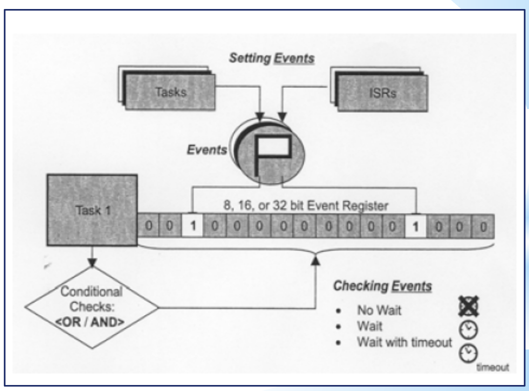
# Event Registers

- Event register is an object belonging to a task (often a part of task's control block)
- A group of binary event flags (8, 16, or 32 bits register)
- A task or an ISR can check or set the flags (bits)
- Used to communicate the occurrence of an event between the tasks and ISR
- Only unidirectional activity synchronization (receiver decides when the synchronization should take place)
  - Sending an event doesn't guarantee state change of receiving task
- Occurrence of the same event can not be counted
- It is an insufficient method for anything other than simple activity synchronization
- Drawback: the source of event not known when multiple sources are possible

29

# Event Register Operations

- Each event is associated with a flag (bit)
- Events in the event register are not queued, events can get lost and the task can not tell which task sent the event
- **Receive** - allows the calling task to receive events from external source with specific conditions such as no wait, timeout, etc.
- **Send** - allows external source (task or ISR) to send events to another task.




30

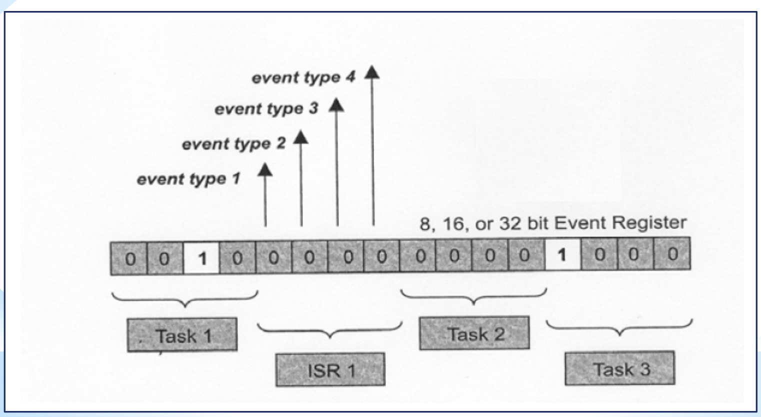
## Event Register Control Block

- **Wanted events** – set of events that the task wishes to receive and are maintained in wanted event register
- **Received events** – these are arrived events and are kept in received event register
- **Timeout value** – the time that the task wishes to wait for arrival of certain event
- **Notification condition** – the task directs the kernel of when it wants to be notified of arrival of an event
  - example: when event 1 and 3 have arrived)

31

## Event Register Typical Use

- The events can be divided into groups in the event register
- The subsets can be associated with a known source and event type based on the relative bit position




32

## VxWorks Events

- *STATUS eventSend (taskId, events)*
  - *taskId:* Task to which the events will be sent
  - *Events:* Events to send
- *STATUS eventReceive (events, options, timeout,*pEventsReceived)*
  - *events:* Events which task wants to request
  - *options:* User options
  - *timeout:* Maximum time to wait for event
    - (clock ticks, WAIT_FOREVER, or NO_WAIT)
  - **pEventsReceived:* Location of the task's event register
- *Show (tId):* allows for viewing of task events

33

## Summary

- Operating system objects are used to facilitate operation of application programs
- Pipes provide unidirectional unstructured data exchange between tasks
- Event registers can be used to communicate events
- Signals are another OS object that are software interrupts
  - we talk about them in later lectures

34

# Assignment for Next Week

- Teams 1- 3: Read and be prepared to present Chapter 12
- Teams 4 -6: Read and be prepared to present Chapter 13
- Lab Report # 3 due at beginning of class

35

# vxWorks Events in-class lab

36