Design & Implementation of Digital Control Systems

Outline

- Example Real-Time Systems
- Hardware for real-time control systems
- Design Decisions
- Implementing Real-Time Behavior
- Task decomposition of control software

Example Real-Time Control Systems

Many real-time systems are real-time control systems.

These systems typically involve:

- control algorithms
- process presentation & display
- operator communication
- data communication
- a mix of continuous and discrete dynamics

Types of real-time control systems:

1. Industrial Control Systems: separate distributed and/or hierarchical control, e.g. process control, manufacturing.
2. Embedded Systems: dedicated controller is part of system, e.g. aerospace, robots, automotive.

Simple Process Control: Buffer Tank

Raw material buffer + heating

Goals:

- Level control: open V when level below L_0, keep the valve open until level above L_1

- Temperature control: PI- controller

Simple Process Control: Buffer Tank

Characteristics:

- Concurrent activities:
  - fluid level control
  - fluid temperature control
- Hard Timing Requirements
- Analog (temperature) & Digital (level) input sensors
- Analog (heating coil current) & Digital (Valve On/Off) actuator outputs
- Continuous (time driven) control and Discrete Event driven control

Electro-Mechanical Control: Active Magnetic Bearings

Ref: RTiC Lab Manual in course pack
http://www.revolve.com

Control position of rotating shaft using electromagnets to provide variable attractive force.

Problem: As the air gap between rotor containing shaft and (fixed) magnet (actuator) decreases, the attractive forces increase, i.e., electromagnets are inherently unstable.

A control system is needed to regulate the current and provide stability of the forces, and therefore, position of the rotor.

Electro-Mechanical Control: Active Magnetic Bearings

Computational tasks:

0. emergency stop

1. periodic fixed rate closed loop suspension (control) loops

2. spin rate measuring system

3. open loop balancing controller

4. data transfer & plotting

5. network transfer tasks

6. miscellaneous additional tasks, e.g. screen refresh, diagnostics

NOTE: AMB systems usually have mechanical backup

Watchdog Control: Reactor Shutdown System (SDS)

What is an SDS?

- control system that monitors reactor parameters
- shuts down (trips) reactor if it observes "bad" behavior
- process control is performed a separate Digital Control computer (DCC) - not as critical

SDS Safety/Performance Considerations:

- Check for short circuits/sensor failures
- Use dead-band to eliminate "chatter"
- Power dependent set points increase operating margin
- "Condition out" sensor in unreliable operating region
- Digital trip output uses "-ve logic" (fail-safe in power loss)

Additional SDS Considerations:

- Use multiple sensors to improve reliability
- Some use of redundant variables to reduce risk of memory errors

Hard real-time requirements:

- different reactor trips must check inputs & respond in as little as approximately 70 msec
- other slower dynamics require monitoring of input sequences over 2-5 seconds.

Soft real-time requirements:

- send data to and receive commands from operator display/interface computer
- data-logging to external system via serial communication
- self-checks

Hardware for real-time control

Control systems typically need:

i) analog & digital I/O hardware & sensors

ii) computational engine(s)

- can use analog sensors with additional hardware to perform A/D
- Analog to digital (A/D) converters often uses 10, 12 or even 16 bits
- encoders can have up to 32-bit accuracy
- Use relays for digital output

Computational engines:

1. DSPs & micro-controllers
2. embedded boards
3. Programmable Logic Controllers (PLCs)
4. Standard PCs with I/O cards

Design Decisions

Hardware & Software partitioning of control systems

Cost: High volume vs. Low volume product

Complexity: Is it a large distributed plant or computationally intensive controller? Ensure adequate resources available but coordination increases complexity

Reliability: What are uptime requirements & operating environment? May need redundancy in hardwre & software

Verifiability: Does system need to be formally verified for regulatory requirement? Isolate safety critical functionality & keep it simple.

Design Revisions

Over their development and life-cycle control systems often go through many revisions:

Implementation Revisions: upgrading to faster computational engines or I/O components

Structural Revisions: include changing

- controller PI to PID or statespace
- sampling style: clock driven vs. event driven
- adding new features, e.g., new trip being added to SDS

Parameter Revision: e.g. tuning parameters in a PID controller

Design Decisions: Hardware & Software Partitioning

Real-time systems are reactive, responding to events. Events may occur at the same time (concurrency).

Work done to service an event is called the *task associated with the event*.

Generally it is good design practice to handle different tasks independently in design.

Example: Buffer Tank

Temperature Loop

LOOP
 measure temperature;
 calculate temperature error;
 calculate heater PI-control;
 output the heater signal;
 wait for h seconds;
END

Level Loop

LOOP
 wait until level below L_0;
 open inlet valve;
 wait until level above L_1;
 close inlet valve;
END;

Design Decisions: Hardware & Software Partitioning

Programming paradigms:

Sequential: Single CPU with manual interleaving of tasks in a cyclic executive

Parallel: Multiple CPUs each running one task

Multi-tasking: Concurrent tasks running on a single shared CPU that switches between processes

Multi-processing: parallel programming on multiple compute engines, each of which might be sequential or multi-tasking

Manual Interleaving (bad way)

```
LOOP
  WHILE level above L0 DO
    measure temperature;
    calculate temperature error;
    calculate heater PI-control;
    output the heater signal;
    wait for h seconds;
  END;
  open inlet valve;
  WHILE level below L1 DO
    measure temperature;
    calculate temperature error;
    calculate heater PI-control;
    output the heater signal;
    wait for h seconds;
  END;
  close inlet valve;
END
```

Complex (brutal) spaghetti code.

Manual Interleaving (better way)

i) Define module to take care of each task

ii) Call update function for each task from within main loop

Example: Buffer Tank

Main Loop:

```
LOOP
  update_temperature_control();
  update_level_control();
  wait for h seconds;
END;
```

Manual Interleaving (cont.)

Temperature Module

```
update_temperature_control()
  BEGIN
    measure temperature;
    calculate temperature error;
    calculate heater PI-control;
    output the heater signal;
  END;
```

Level Module

```
update_level_control()
  BEGIN
    IF level below L0 THEN
      open inlet valve;
    ELSIF level above L1 then
      open inlet valve;
    END;
  END;
```

Manual Interleaving (cont.)

For more complex scheduling to meet time constraints can change relative frequeny of update functions within main loop. E.g.,

Main Loop:

```
LOOP
  update_task_A();
  update_task_B();
  update_task_A();
  update_task_C();
  update_task_B();
  update_task_A();
  wait for h seconds;
END;
```

Problem: Main loop timing depedent upon timing of each task. A change in one module can force redesign of main loop to meet timing constraints.

Solution: Let RTOS schedule tasks.

Implementing Real-Time Behavior

Real-time behavior can be achieved by:

- using a stand-alone (single loop) program
- faking it by being fast
- using timer interrupts
- writing a real-time kernel
- using an existing (uncertified) real-time operating system

What do we mean by each of these choices?

When it is appropriate to implement real-time behavior by each of the above?

What are the advantages and disadvantages?

Stand-alone Single Loop Program

This just corresponds to the manual interleaving case already described.

Advantages:

- low overhead - no resources required for OS
- easier to verify since only relies on your code

Disadvantages:

- Difficult to get timing right using manual interleaving when there are multiple tasks
- Change in timing of one task affects overall timing of main loop, possibly forcing redesign on pre-run schedule

Faking it by Being Fast

It is possible to implement some limit RT behavior using a non-RT OS such as Linux or NT.

How?

1. Use interrupt service routines to process events

2. Use OS's scheduler

With Linux or NT can achieve average response on order of ms but process can be blocked by system activity (e.g., mouse or keyboard input, network activity, etc).

Implementing Periodic Tasks

Using an RTOS with real-time API to implement general periodic behavior.

Attempt 1:

```
LOOP
  PeriodicActivity;
  rt_sleep(h);
END;
```

Does not work.

Period $> h$ and time-varying.

The execution time of PeriodicActivity is not accounted for.

Implementing Periodic Tasks

Attempt 2:

```
LOOP
  Start=rt_get_time();
  PeriodicActivity;
  Stop=rt_get_time();
  elapsed := Stop - Start;
  rt_sleep(h-elapsed);
END;
```

Does not work. An interrupt causing suspension may occur between the assignment and rt_sleep().

In general, an rt_sleep(Delay) primitive is not enough to implement periodic processes correctly.

A rt_sleep_until(DelayUntil) primitive is needed.

Implementing Periodic Tasks

Attempt 3:

```
LOOP
  t=rt_get_time();
  PeriodicActivity;
  t=IncTime(t,h);
  rt_sleep_until(t);
END;
```

Does not work. An interrupt may occur between the rt_sleep_until(t) and rt_get_time().

Implementing Periodic Tasks

Attempt 4:

```
t=rt_get_time();
LOOP
  PeriodicActivity;
  t=IncTime(t,h);
  rt_sleep_until(t);
END;
```

Best solution using "one shot" timers.

Will try to catch up if the actual execution time of PeriodicActivity occasionally becomes larger than the period.

Implementing Periodic Tasks

Implementing periodic behavior with "one-shot" timers is inefficient.

Typically requires 3× longer setup time than using periodic timer.

Solution: Make the task periodic and allow RT-scheduler to schedule task using its periodic timers.

Example: Generate a 10kHz square wave on a pin of the parallel port.

The variation in time of execution of a periodic process is known as *jitter*, i.e.,

$$\text{jitter} = \begin{array}{c}\text{time of expected}\\ \text{execution start}\end{array} - \begin{array}{c}\text{actual time}\\ \text{execution begins}\end{array}$$

RTOS try to minimize jitter.

The Real-Time Linux API

The RT-Linux function calls used in the 10kHz square wave program are:

`rt_get_time` returns the time in ticks .

`rtl_task_init` sets up, but does not schedule a task.

`rt_task_make_periodic` asks the periodic scheduler to the run task at a fixed period (given as a parameter).

`rt_task_wait` yields the processor until the next time slice for this task.

Concurrent Programming

What happens when there is more than one task we want to run?

1. *Multiprogramming:* the processes multiplex their execution on a single CPU

2. *Multiprocessing:* the processes multiplex their execution on a multiprocessor system with tightly coupled processors

3. *Distributed processing:* the processes multiplex their

There is (some) true parallelism in (2) and (3).

Often we have a single CPU (1) or fewer CPUs than tasks (2), so we need to "fake" concurrent processes via interleaving semantics. This is known as *logical parallelism*.

RTOS Basics

A Real-Time Operating System (RTOS) typically consists of

- a sequential language (C) with real-time primitives (the real-time API)
- real-time kernel for process handling

The *Kernel* is the part of multitasking system responsible for management of CPU time (scheduling) and providing facilities for interprocess coordination, i.e.

- management of the tasks
- communication between tasks

When the kernel decides to run a different task, it simply saves the Task's Context (CPU registers) in the current tasks storage area.

Context Switching

In a *preemtive kernel* the highest priority task ready to run is always given the CPU. A context switch happens when:

- the running process voluntarily releases the CPU (e.g., process calls `rt_task_wait`)
- when the running process performs an operation that causes a blocked process of higher priority to become ready
- when an Interrupt Service Routine has occurred which causes a blocked process of higher priority to become ready

With a preemptive kernel, execution of the highest priority task is deterministic (i.e., it can be determined when that task will get control of the CPU).

Context Switching (cont.)

What happens during a context switch?

1. **Disable** all interrupts
2. **Save** the state of the processor immediately before the switch is saved on the stack of the suspended process
3. **Switch** the context:
   - the value of the stack pointer is stored in the process record of the suspended process
   - the stack pointer is set to the value that is stored in the process record of the new process
4. **Restore** the state of the resumed process is restored (popped from its stack)
5. **Renable** interrputs

The time required to switch from one task to another is the *context switching time*.

Interrupts

(see class notes)

- interrupt latency

- interrupt service routine

- interrupt priority level

- kernel

- kernel module

What's a Priority?

How does the kernel decide which process has the highest priority?

Each process is assigned a number that reflects the importance of its time demands. Typically a lower number means a higher priority i.e.,

"The Number 1 priority is . . . "

RT-Linux starts priorities at 1 though some other RTOS including the RTAI version of Linux start priorities at 0. Both real-time versions of linx have a lowest priority `RT_LOWEST_PRIORITY`.

Using priority-based scheduling, processes that are Ready to execute are stored in task queue according to priority.

Static vs. Dynamic Priorities

With (*static*) priority-based scheduling the priority of a process is fixed. This is known as *Fixed priority scheduling*.

There are different methods of having *dynamic* priorities that change depending upon time and circumstance. One common dynamic priority based scheduling policy is familiar to Engineering students: *Earliest Deadline First (EDF) scheduling*.

In EDF scheduling the closeness of the deadline for completion of a task determines its priority.

RT-Linux Fixed Priority Scheduler

How is a task chosen to preempt a running task?

In one-shot timer mode, if there are no periodic tasks it uses:

```
inline static struct rtl_thread_struct *
find_preemptor(schedule_t *s,
               struct rtl_thread_struct *chosen){
  struct rtl_thread_struct *t;
  struct rtl_thread_struct *preemptor=0;
  for (t = s->rtl_tasks; t; t = t->next) {
    if ( t->state == RTL_THREAD_DELAYED )
    {
      if (t->sched_param.sched_priority >
          chosen->sched_param.sched_priority)
      {
        if(!preemptor ||
           (t->resume_time < preemptor->resume_time))
        {
          preemptor = t;
        }
      }
    }
  }
  return preemptor;
}
```

Scheduling Theory

How do we maximize use of resources & determine if we can meet all deadlines?

Problem: Sometimes these goals conflict.

E.g., to maximize throughput (number of tasks run in a given amount of time) a multitasking OS such as Linux or NT might try to minimize the:

- (weighted) sum of completion times
- schedule length
- number of processes required

These metrics are not useful for real time systems since there is no direct assesment of timing properties.

Scheduling Theory (cont.)

Minimizing the sum of completion times is useless if it causes large jitter in a periodic process or forces task to be completed after its deadline.

We need some real-time scheduling policies that are "optimal". Lets consider the following definition of optimal:

> *A scheduling algorithm is* optimal *if, when it fails to meet a deadline then no other algorithm can meet it.*

EDF Dynamic Scheduler

States of EDF task model (from its `rt_sched.h`):

```
enum {RT_TASK_READY, RT_TASK_DELAYED, RT_TASK_DORMANT};

struct rt_task_struct {
        int *stack;     /* hardcoded */
        int uses_fp;    /* this one is too   */
        int magic;
        int state;
        int *stack_bottom;
        int priority;
        int identifier;              /* EDF */
        RTIME period;
        RTIME resume_time;
        RTIME relative_deadline;     /* EDF */
        RTIME absolute_deadline;     /* EDF */
        struct rt_task_struct *next;
};

typedef struct rt_task_struct RT_TASK;
```

Note: Task also has an implicit state of "`RT_TASK_RUNNING`" when it is the task pointed to by variable `rt_current`.

EDF Dynamic Scheduler (cont)

The EDF scheduler is:

```
void rt_schedule(void)
{
    RTIME now;
    RTIME preemption_time;
    RT_TASK *task;
    RT_TASK *new_task;
    RT_TASK *preemptor;
    int flags;
    RTIME next_deadline;        /* EDF */

    r_save_flags(flags);
    r_cli();
    now = rt_get_time();

    for (task = rt_tasks; task; task = task->next) {
        if (task->state == RT_TASK_DELAYED &&
                    task->resume_time < now + 10) {
            task->state = RT_TASK_READY;
        }
    }
```

Above `for` loop implements Task model transitions :

$$\texttt{RT\_TASK\_DELAYED} \rightarrow \texttt{RT\_TASK\_READY}$$

EDF Dynamic Scheduler (cont)

Core of the EDF algortithm is:

```
    new_task = &rt_linux_task;

/* BEGIN EDF ************************************/
    next_deadline = rt_linux_task.absolute_deadline;

    for (task = rt_tasks; task; task = task->next) {
      if (task->state == RT_TASK_READY &&
        task->absolute_deadline < next_deadline) {
      new_task = task;
      next_deadline = task->absolute_deadline;
      }
    }

    preemptor = 0;
    preemption_time = RT_TIME_END;
    for (task = rt_tasks; task; task = task->next) {
      if (task->state == RT_TASK_DELAYED &&
        task->resume_time < preemption_time) {
      preemption_time = task->resume_time;
      preemptor = task;
      }
    }
/* END EDF **************************************/
```

1st `for` loop determines ready task with earliest deadline.

2nd loop determines when next task will become ready and we'll have to reschedule again.

EDF Dynamic Scheduler (cont)

```
    if (preemptor) {
        rt_set_timer(preemption_time);
    } else {
        rt_no_timer();
    }

    if (new_task == rt_current) {
      r_restore_flags(flags);
        return;
    }

    if (new_task == &rt_linux_task) {
        SFIF = linux_irq_state;
    } else if (rt_current == &rt_linux_task) {
        linux_irq_state = SFIF;
        SFIF = 0;
    }

    new_task->state = RT_TASK_READY;
    rt_switch_to(new_task);
    r_restore_flags(flags);
}
```

For the ready task with earliest deadline `rt_switch_to(new_task)` implements transition:

$$\texttt{RT\_TASK\_READY} \rightarrow \texttt{RT\_TASK\_RUNNING}$$

Rate Monotonic (RM) Scheduler

Core of the RM algortithm is:

```
    new_task = &rt_linux_task;

/* begin rate-monotonic scheduling  */
    shortest_period = rt_linux_task.period;

    for (task = rt_tasks; task; task = task->next) {
      if (task->state == RT_TASK_READY &&
        task->period < shortest_period) {
        new_task = task;
        shortest_period = task->period;
      }
    }

    preemptor = 0;
    preemption_time = RT_TIME_END;
    for (task = rt_tasks; task; task = task->next) {
      if (task->state == RT_TASK_DELAYED &&
        task->resume_time < preemption_time) {
        preemption_time = task->resume_time;
        preemptor = task;
      }
    }
/* End of rate-monotonic scheduler */
```

1st `for` loop determines ready task with shortest period

2nd loop determines when next task will become ready and we'll have to reschedule again.