CT420 REAL-TIME SYSTEMS

CYCLIC EXECUTIVE SCHEDULING

Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

Lecture Overview

- Overview RTS scheduling approaches
- Cyclic Executive Approach
- Dealing with Interrupts

Recap: Quality Requirements for RTSCS

- RTSCS must be **time responsive**
- RTSCS must be **reliable**
  - The ability to behave in accordance with its specification
- RTSCS must be **safe**
  - Conditions that lead to hazards do not occur
- RTSCS must be **secure**
  - Protect itself against intentional or accidental access, use, modification or destruction
- RTSCS must be **usable**
  - Easy to learn, understand, and use
- RTSCS must be **maintainable**
  - Return swiftly to an operational state after receiving repairs or modification (e.g. plug in-and-forget)

Which Programming Languages are (not) suitable to implement hard RTS?

- **Unsuitable programming languages include:**
  - **Java, Python, Ruby, JS**
    - Their garbage collection can introduce non-deterministic behavior, as it can pause the execution of the program at unpredictable times, causing delays
    - Dynamic typing can lead to unpredictable performance
- **Suitable programming languages include:**
  - **C, C++, Ada, Real-Time Java**
  - Using C as an implementation language we now look into implementing synchronous tasks, starting with the **cyclic executive approach**

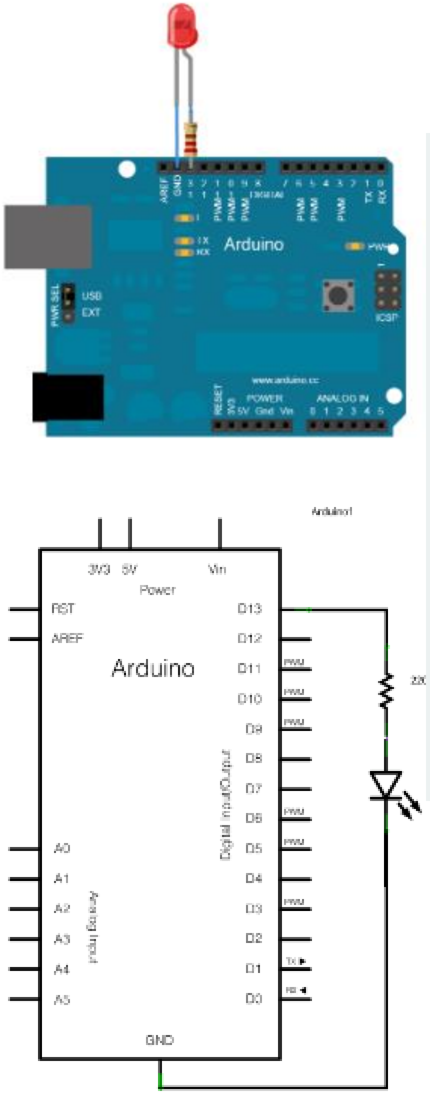
Cyclic Executive Approach

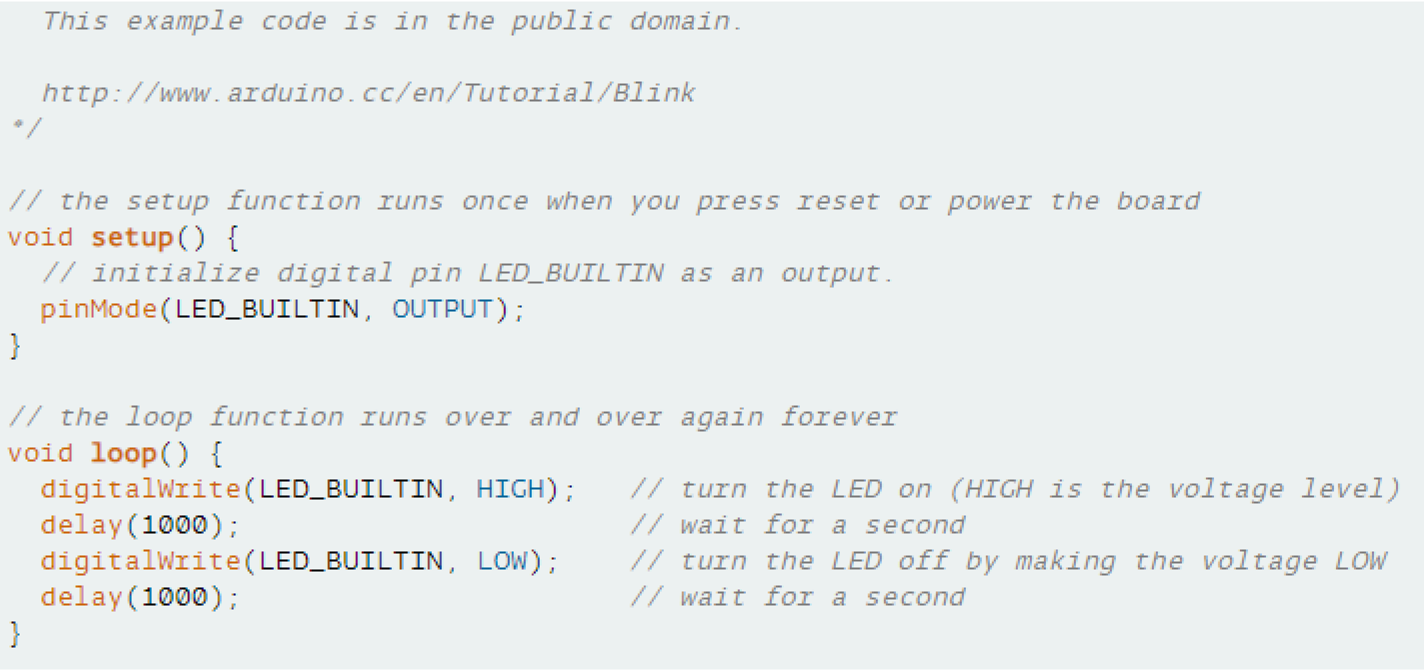
- Single Process

```
while(1){
  Task 1;
  Task 2;
  ..
  ..
  Task n;
}
```

- No Operating System → No scheduler
- Manually construct cycle schedule
- Encapsulate all tasks within single infinite loop
- Tasks in this context are simply functions with or without function arguments

Trivial Arduino Example: Blinking LED





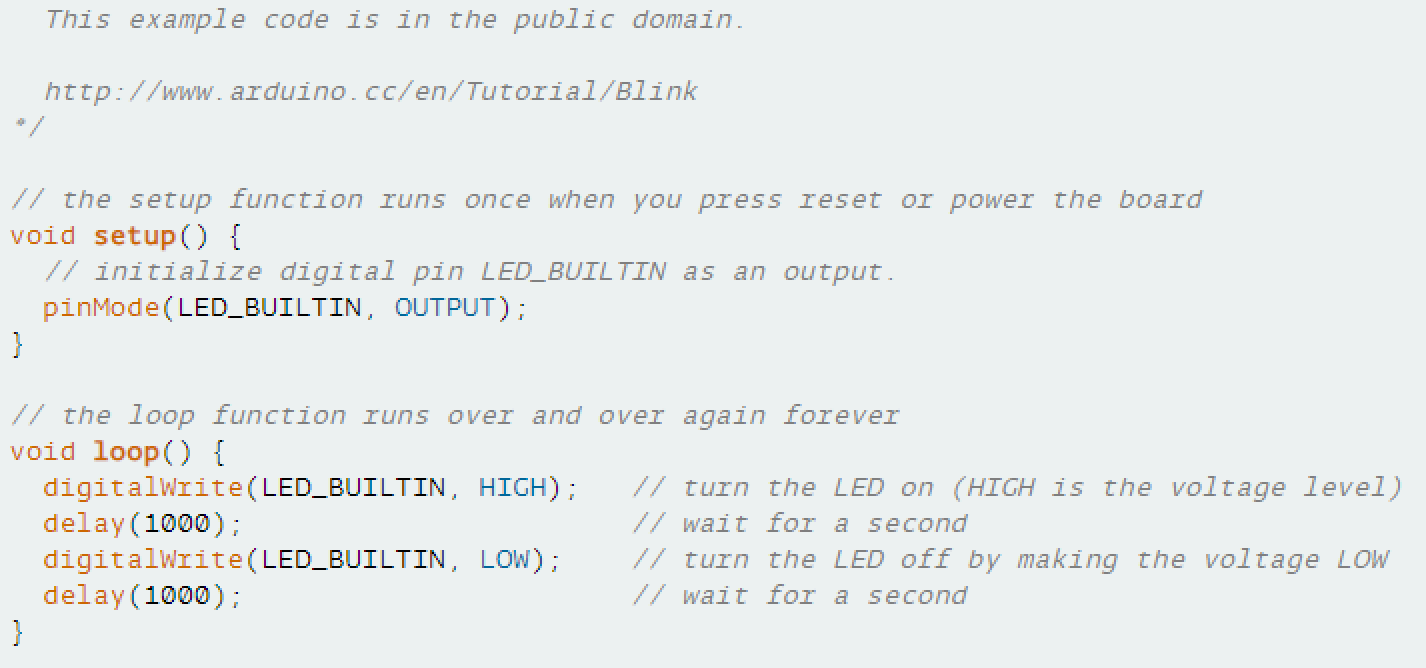
```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Example for poorly programmed Scheduler



```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

1 ms →
1000 ms →
1 ms →
1000 ms →

- Total execution time per loop: 2002 ms ☹
- Also, the execution time of digitalWrite() could vary, resulting in variable loop execution times
- Therefore, we need to consider a better approach to program such schedulers

Cyclic Executive

- Used for very well defined / periodic tasks with bounded execution times
- Need to ensure that tasks cannot block and halt all others
- Tasks may run at different frequencies
- E.g. they control different parameters with different physical characteristics (like temperature, pressure, voltage in power station)
- Overall cycle either
  - will run as fast as processor can handle tasks
  - is slowed down by delay() function, i.e. may need to slow tasks down to meet particular RTS requirements (e.g. measure steam pressure every 50 ms)

Cyclic Executive

- Used for very well defined / periodic tasks with bounded execution times
- Need to ensure that tasks cannot block others
- Tasks may run at different frequencies
  - Possible Strategies
    - Run as fast as required by highest frequency task
    - Use lower harmonics for remaining tasks
    - Possible use of counters to control sequence
  - Use of major and minor cycles
    - E.g. Highest frequency task is 100 Hz
      - Other tasks at 50Hz, 25 Hz, etc.
    - Use of timers/interval timers (rather than delay() function) to correctly 'schedule' tasks → different to Arduino example

Cyclic Executive

- Task Set
  - Major Cycle = 40 Hz
  - Minor Cycle = 10 Hz
- Use interval timer interrupts to enable scheduler to loop through minor cycles
- Manually construct schedule to meet criteria
- Note: For now we just assume that task exec times are bounded!

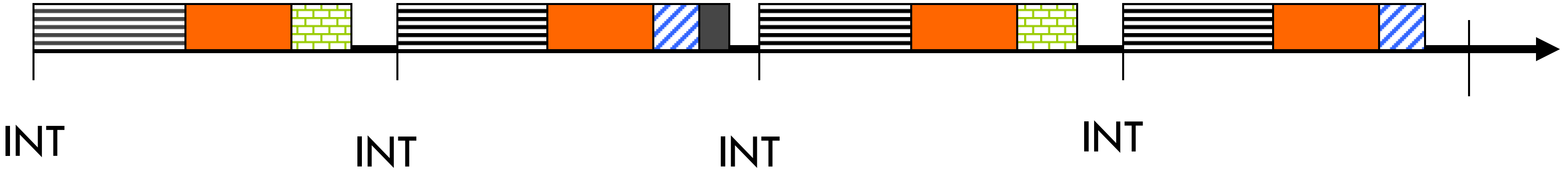
Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

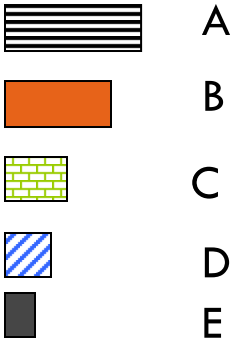
	Execution time	Deadlines	Software size	Software complexity
Hard - Fast	●●●●	●●●●	●	●
Hard - Slow	●	●●●●	● → ●●●	● → ●●●●
Soft - Fast	●●●●	●●	● → ●●●	● → ●●●
Soft - Slow	●●	●●	● → ●●●●	● → ●●●●

Attribute rating
● Low ●●●● high

CE Time Line

Interrupts generated every 25 ms via an interval timer
→ Tasks are launched **every** 25 ms (different to Arduino example)





Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

CE Pseudocode

```
loop
wait_for_INT
   task_A
   task_B
   task_C
wait_for_INT
   task_A
   task_B
   task_D
   task_E
wait_for_INT
   task_A
   task_B
   task_C
wait_for_INT
   task_A
   task_B
   task_D
end loop
```

```
volatile int timerFlag = 0;
...
void wait_for_INT() {
   while timerFlag == 0) {}
   timerFlag = 0;
}

void interrupt timer_ISR() {
   timerFlag = 1;
}
```

Case Study NAS-Box

- Enclosure that provides space, power and control options for many (12- 60) hard disks (Network Attached Storage)
- Enclosure controller must handle a few crucial management tasks, including optimised temperature control (overheating of disks) while minimising cooling fan noise emissions

Example NAS Box Controller CE

#	Task	Period p [ms]	Exec Time [ms]
1	X = ReadTempSensorA()	70	10
2	Y = ReadTempSensorB()	70	10
3	Z = Voter(X, Y)	70	5
4	SetFan(Z)	70	15
5	CheckDrives()	140	20
6	SetDriveLeds()	140	5
7	SelfTest()	140	15

Task Dependencies (i.e. tasks are simply ordered):

- #1 + #2 → #3 → #4
- #5 → #6

In-Class Activity

```
loop
wait_for_INT
    A…
wait_for_INT
    B…
end loop
```

Tasks:

1. Construct a suitable CE for the SAN example, i.e. Determine the sequence of tasks for **A** and **B** (e.g. “A125B213”)

1. Determine timer settings, i.e. how often is the timer ISR invoked?

CE Example – My Solution

```
loop
wait_for_INT (70 ms)
   #1 (10 ms)
   #2 (10 ms)
   #3 (05 ms)
   #4 (15 ms)
   #5 (20 ms)
   #6 (05 ms)
wait_for_INT (70ms)
   #1 (10 ms)
   #2 (10 ms)
   #3 (05 ms)
   #4 (15 ms)
   #7 (15 ms)
end loop
```

Tasks:

1. Construct a suitable CE for the SAN example
2. Determine timer settings, i.e. how often is the timer ISR invoked?

Example Code

```
volatile int timerFlag;
main() {
        int X, Y, Z, state = 0;
        StartTimer();
        while (1) {
                 timerFlag = 0;
                X = ReadTempSensorA();
                Y = ReadTempSensorB();
                Z = Voter(X, Y);
                SetFan(Z);

                switch(state) {
                case 0:
                                SelfTest();
                                break;
                case 1:
                                CheckDrives();
                                SetDriveLeds();
                                break;
```

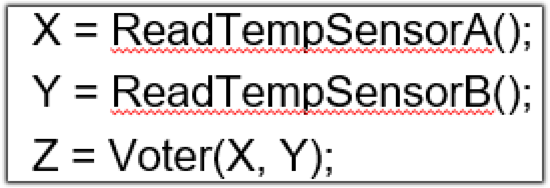
```
                default:
                                break;
                }
                if (timerFlag ==1) {
                                TimeoutError(state);
                }
                else {
                                while (timerFlag == 0) ;
                }
                state ^= 1;
        }
}
void interrupt TimerISR() {
        timerFlag = 1;
}
void StartTimer(void) {
        /* Set hardware timer to 70 ms interval. */
        /* ... */
}
```

Keyword *volatile*

- ***volatile*** is a qualifier that is applied to a variable when it is declared
- It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby
- Why is ***timerFlag*** in the previous example a volatile variable?

Cyclic Executive

- Major cycle must be multiple of minor cycle
- All tasks share common address space
  - Can pass data easily
  - Little/no need for data protection (e.g. via semaphores / mutex)
    - Only one task operates at any time
    - → No concurrent access possible



```
X = ReadTempSensorA();
Y = ReadTempSensorB();
Z = Voter(X, Y);
```

- Large tasks may need to be subdivided to facilitate/meet overall schedule → adds to complexity
- Inflexible
  - Adding a new task may involve a lot of work
  - Hardware specific → very limited portability

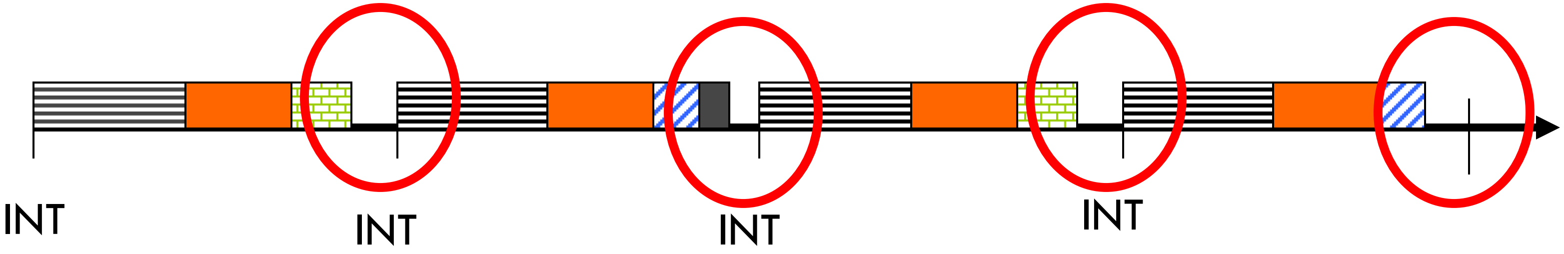
Cyclic Executive and ISRs

- Management of **asynchronous** events (interrupts) tricky
- Only workable, if:
  - ISR is decoupled from other tasks (no dependencies like blocking)
  - Maximum number of ISR executions does not cause task overrun

Task	**Period p [ms]**	**Exec Time [ms]**
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

CE Time Line: Slack for asynchronous Interrupts

Synchronous interrupts generated every 25 ms, e.g. via interval timer



▤ A

🟧 B

▦ C

▨ D

⬛ E

Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

Example Code with Overrun Check

```
volatile int timerFlag;
main() {
    int X, Y, Z, state = 0;
    StartTimer();
    while (1) {
         timerflag = 0;
        X = ReadTempSensorA();
        Y = ReadTempSensorB();
        Z = Voter(X, Y);
        SetFan(Z);

        switch(state) {
        case 0:
            SelfTest();
            break;
        case 1:
            CheckDrives();
            SetDriveLeds();
            break;
        default:
            break;
        }
        if (Timerflag ==1) {
            TimeoutError(state);
        }
        else {
            while (TimerFlag == 0) ;
        }
        state ^= 1;
    }
}
void interrupt TimerISR() {
    timerFlag = 1;
}
void StartTimer(void) {
    /* Set hardware timer to 70 ms interval. */
    /* ... */
}
```

Example Code with Overrun Check and ISR Limitation

- The previous example allows detecting task overruns, using the *TimerFlag* variable
- However, it does not prevent it from happening, i.e. the scheduler gives ISR execution priority over timeliness of tasks
- Alternatively, one can give timeliness of task execution over ISR execution, assuming that limiting the number of ISR calls doesn't break the system
    - I.e. it's a bad idea if all events/ISRs have an impact on system safety (→ airbag deployment)
- In the next example, *intCounter* represents the CPU time used for ISR execution, while *ISR_LONG* and *ISR_SHORT* are an upper boundary for the ISR execution times; the total sum of these over a single cycle must not exceed *MAX_INTCOUNT*

Example Code with Overrun Check and ISR Limitation

```
#define MAX_INTCOUNT    4 // Max value for ISR execution times
#define ISR_LONG        4 // Relative execution time of ISR
#define ISR_SHORT       2 // Relative execution time of ISR
volatile int timerFlag, intCounter;
main() {
    int X, Y, Z, state = 0;
    StartTimer();
    while (1) {
          timerflag = 0;
          intCounter = 0;
        X = ReadTempSensorA();
        Y = ReadTempSensorB();
        Z = Voter(X, Y);
        SetFan(Z);

        switch(state) {
        case 0:
                SelfTest();
                break;
        case 1:
                CheckDrives();
                SetDriveLeds();
                break;
            default:
                    break;
            }
            if (Timerflag ==1) {
                    TimeoutError(state);
            }
            else {
                    while (TimerFlag == 0) ;
            }
            state ^= 1;
        }
}
/* void interrupt TimerISR() and void StartTimer(void) as seen before */
void interrupt OtherISRLong(void){
        if (intCounter + ISR_LONG > MAX_INTCOUNT) {
            /* Do nothing and exit */ } else {
             intCounter += ISR_LONG;
            /* Execute ISR code and exit */
            ...                              }
}
void interrupt OtherISRShort(void){
        if (intCounter + ISR_SHORT <= MAX_INTCOUNT) {
             intCounter += ISR_SHORT;
            // Execute ISR code and exit
            ...          }
        }
```

Next Topic: Benchmarking and WCET

Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- Principal question: How do we determine (worst case) execution times of tasks?