

# Embedded systems engineering

David Kendall

Northumbria University

- What's wrong with *super-loop*?
- What is a time-triggered scheduler?
- How to construct a schedule?
- What are the pros and cons?
- How can I implement a time-triggered scheduler in practice?
  - Based on Pont but for ARM target and with some restructuring of code.

# Super loop architecture

```
/* My embedded system as a super loop */  
  
init();                /* Prepare to start          */  
  
while (true) {          /* repeat forever          */  
    f();                /* execute the control function */  
}
```

- Pros

- Simple – easy to understand
- Uses almost no resources

- Cons

- Difficult to ensure that  $\tau()$  is called at precise instants of time.
- Many embedded systems require precise timing
  - Periodic tasks
  - One-shot tasks

# Can we fix super loop?

```
/* My embedded system as a super loop */  
  
init ();                                /* Prepare to start */  
  
while (true) {                          /* repeat forever */  
    f ();                               /* execute the control function */  
    delay(n);                          /* delay for n microseconds */  
}
```

- This might work – repeat  $f()$  every  $m + n$  microseconds, where  $m$  is the execution time of  $f()$
- But
  - to choose  $n$  we need to know  $m$  precisely
  - execution time of  $f()$  must be the same each time round the loop
- Unrealistic assumptions

# Fix number 2

```
/* My embedded system as a super loop */  
  
init();  
  
while (true) {  
    start = getCurrentTime();  
    f();  
    delay(start + p - getCurrentTime());  
}
```

- This is better
  - repeat  $f()$  every  $p$  microseconds (more or less)
  - time for  $f()$  can vary on each iteration but period remains constant
- But ...
  - Need to allow for time taken to get the time and configure the delay
  - Difficult to break controller into multiple functions that can execute at different rates

# Towards a better solution

- Use timer-based interrupts to ensure that functions are called at precise instants of time.
- For example, ...

```
void f(void);                /* Control function prototype */

int main () {

    initTimer(TIMER0, f, 10); /* Set timer to interrupt at 10Hz */
    while (true) {
    }
}

void f(void) {                /* Control function – interrupt handler */

    ledToggle(USB_LINK_LED);

    /* clear interrupt */
    clearInterruptTimer(TIMER0);
}
```

# Executing multiple tasks at different time intervals

- Embedded system may consist of multiple tasks that need to execute at different time intervals, e.g.
  - Read input from an ADC every millisecond
  - Read one or more switches every 200 milliseconds
  - Update LCD display every 3 milliseconds
- How to solve this problem?
  - Use multiple timers?
    - No - why not? ...coming next
  - Use a time-triggered scheduler?
    - Yes

# Why not use multiple timers?

- May not have enough timers
  - e.g. 100 tasks into 4 timers does not go
- Code becomes hard to maintain
  - e.g. Change of oscillator frequency may involve modification to all tasks
  - e.g. adding another task not be possible if all timers are currently used
- Need to handle simultaneous interrupts
  - difficult to manage, hard to predict behaviour
  - system much simpler if there's only a single interrupt source



# What is a time-triggered scheduler?

- extraordinarily simple operating system that allows tasks to be called on periodic and/or one-shot basis
- single timer ISR shared by many tasks, so
  - only one timer needs to be initialised
  - changes of timing source require only local code changes – usually one function at most
  - same scheduler can be used no matter how many tasks
- The time-triggered scheduler relies on a *static* schedule for its correct operation.

# Static and Dynamic Scheduling

## Static scheduling

In the *static scheduling* approach, all decisions about which task should run at any given time are made *offline*, i.e. *before* run-time. The job of the scheduler at run-time is very simple: it consults a scheduling table to see which task should run next and runs it. Typically, execution of a task is *non-preemptive*, i.e. it *runs to completion*.

## Dynamic scheduling

In the *dynamic scheduling* approach, decisions about which task should run are made *online*, i.e. at run-time. The job of the scheduler is to determine which task should run next, according to some criteria, and then run it. Typically, execution of a task is *preemptive*. Examples of dynamic scheduling algorithms include fixed priority preemptive algorithms such as rate monotonic and deadline monotonic, and dynamic priority algorithms such as earliest deadline first and least laxity.

# Periodic task model

- We assume that the job of the time-triggered scheduler is to run a set of periodic and one-shot tasks at pre-defined times.
- The periodic tasks are characterised by their
  - **period** ( $p$ )
  - **phase** also known as *offset*, ( $\phi$ )
  - **worst-case execution time** ( $e$ )
  - **deadline** ( $d$ )
- We assume a system of  $N$  tasks comprises an indexed set of periodic tasks  $T$

$$T = \{T_i : i \in 1..N\}$$

- Each periodic task can be regarded as generating *instances* of itself for execution, with instances numbered starting at 0.
- The **arrival time** of the  $j$ th instance of task  $i$  is

$$\alpha(T_{i,j}) = j * p_i + \phi_i$$

- **Harmonic periods** – the periods of a task set are *harmonic* iff every period in the task set is an integer multiple of all smaller periods in the set
- **Hyperperiod** – the *hyperperiod* of a task set is the greatest time that elapses until the pattern of task arrivals is repeated
  - The hyperperiod is equal to the least common multiple (LCM) of the periods of tasks in the set.
  - For a task set with harmonic periods, the hyperperiod is equal to the greatest of the periods of the tasks in the set.
- **Utilisation** – the *utilisation*,  $U$ , of a task set  $T = \{T_i \mid i \in 1..N\}$  is given by

$$U = \sum_{i=1}^N \frac{e_i}{p_i}$$

# Structured time-triggered scheduler

- A time-triggered scheduler can be implemented simply by using a *periodic* timer interrupt.
- The period of the timer interrupt defines a *frame* length
- Several tasks may be scheduled sequentially within a frame
- Let  $Z$  be the frame length and  $H$  be the hyperperiod. Then, the table that drives the scheduler has  $F = \frac{H}{Z}$  entries
- Each entry lists the jobs to be executed in that frame
- The scheduler
  - is called by the timer interrupt
  - determines which frame should be scheduled
  - executes all jobs sequentially in the current frame
- The schedule repeats itself every hyperperiod.

# Requirements for a schedule

- Hyperperiod  $H$  is least common multiple of periods of task set
- Frame size  $Z$  should be an integer divisor of  $H$ , at least as big as  $\max\{e_i\}$  and no bigger than  $\min\{p_i\}$ . Usually  $\gcd\{p_i\}$  is a good choice.
- Every job instance should be scheduled in exactly one frame
- No job instance should be scheduled before its release time
- The sum of the worst case execution times of the job instances scheduled in any frame should be no bigger than the frame size
- The deadline for any job instance should be no earlier than the start of the next frame following the one in which it's scheduled
- These requirements can be expressed formally as an *integer linear program (ILP)* and a schedule can be produced automatically by a solver for task sets of moderate size.

# Linear Program

- A *linear program* has the general form:
  - *maximise* (or *minimise*) an *objective linear function*,  $\mathbf{c}^T \mathbf{x}$
  - *subject to* a set of *linear constraints*,  $A\mathbf{x} \leq \mathbf{b}$
  - and  $\mathbf{x} \geq 0$
  - $\max\{\mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x} \geq 0\}$
- Linear programming is used for *optimisation* in many contexts, e.g. planning, scheduling, finance, etc.
- If, in addition, there is an additional requirement that the variables are integers, then this is an *integer linear programming* (ILP) problem
- If, only some variables are required to be integers, it is a *mixed integer linear programming* (MILP) problem.
- There are efficient solvers for LP, e.g. CPLEX and Gurobi.
- There are higher-level modelling languages, e.g. AMPL, that can be used to give a convenient representation of problems that can then be translated into LPs
- We can use ILP to solve the uniprocessor, non-preemptive scheduling problem for a set of periodic tasks

# Constructing a schedule

```
# Model for the construction of static, non-preemptive, frame-based schedules
# written in AMPL (www.ampl.com)
# DK – 12–10–2014

param H >= 0;                                # hyperperiod
param Z >= 0;                                # frame size
param F = H / Z;                             # number of frames

set FRAME = 0..F-1;                          # indexing set for frames
set TASK;                                    # indexing set for tasks

param phase {TASK} >= 0;                     # task parameters
param period {TASK} >= 0;
param wcet {TASK} >= 0;
param deadline {TASK} >= 0;

set INST {i in TASK} = 0 .. H / period[i] - 1; # indexing set for task instances

var X {i in TASK, INST[i], FRAME} binary;    # X[i,j,k] = 1 if the jth instance of
                                              # task i is scheduled in frame k

minimize makespan:
    sum {i in TASK, j in INST[i], k in FRAME} X[i,j,k] * k * Z;

subject to scheduledExactlyOnce {i in TASK, j in INST[i]}:
    sum {k in FRAME} X[i,j,k] = 1;
subject to notScheduledBeforeRelease {i in TASK, j in INST[i], k in FRAME}:
    X[i,j,k] = 0 or j * period[i] + phase[i] <= k * Z;
subject to fitInFrame {k in FRAME}:
    sum {i in TASK, j in INST[i]} X[i,j,k] * wcet[i] <= Z;
subject to observableCompletionByDeadline {i in TASK, j in INST[i], k in FRAME}:
    X[i,j,k] = 0 or k * Z + Z <= j * period[i] + phase[i] + deadline[i];
```



# Example task sets

## Task set with non-harmonic periods

```
param H := 60;
param Z := 5;
set TASK      := 1 2 3 4;

param: phase  period  wcet  deadline :=
1         0       5      3        5
2         0      10      1       10
3         0      15      1       15
4         0      20      2       20
;
```

## Task set with harmonic periods

```
param H := 80;
param Z := 10;
set TASK      := 1 2 3 4 5;

param: phase  period  wcet  deadline :=
1         0      10      3       10
2         1     20      1       20
3         2     40      1       40
4         3     40      2       40
5         4     80      7       80
;
```

Obtain a demo version of the `ampl` tool from <http://ampl.com/try-ampl/download-a-demo-version/>

Follow the installation instructions for your platform

Construct schedule with:

```
ampl: model schedule.mod;
ampl: data schedule.dat;
ampl: option solver cplex;
ampl: solve;
ampl: display X;
```

- Much of the (early) literature about embedded systems development introduces the ideas of time-triggered scheduling using a different vocabulary.
- The time-triggered scheduler is known as a **cyclic executive**
  - Manually constructed, off-line schedule of periodic tasks (procedure calls)
- Concurrent design, but sequential code (collection of procedures)
- Procedures are mapped onto a sequence of minor cycles (frames)
- Minor cycles constitute the complete schedule: the major cycle (hyperperiod)

# Cyclic executive: properties and requirements

- No actual processes exist at run-time (only procedures)
- Minor cycles are sequences of procedure calls
- All periods must be a multiple of minor cycle time
- General rule:
  - minor cycle time is **gcd** of periods
  - major cycle time is **lcm** of periods
- Procedures share a common address space
  - Useful for inter-"process" communication
  - Only need one stack for user processes
  - No need for memory protection: concurrent access not possible
  - Deadlines are guaranteed by the offline schedule

# Time-triggered scheduler: against – traditional view

- Difficult to incorporate:
  - processes with long periods
    - major cycle time determines maximum period
    - can (sometimes) be (partially) solved with secondary scheduling
  - processes with long computation times: must be split into several procedures
  - processes that are **sporadic** (not periodic but with well-defined minimum inter-arrival time)
- Difficult to construct and maintain the schedule
  - Fixed number of fixed sized procedures required
  - May cut across useful and well-established boundaries
  - Potentially very bad for software engineering (error prone)
- More flexible scheduling methods are difficult to support
- Determinism is an unnecessarily strong property; what is required is **predictability**

# Time-triggered scheduler: case for the defence

- Generally accepted benefits:
  - The scheduler is simpler
  - The overheads are reduced
  - Testing is easier
  - Certification authorities tend to support this form of scheduling
- Most damaging of the problems:
  - Long-running computations either make the system unresponsive (minor cycle time too great) or must be split up artificially
- But Michael Pont [PON10, chp 13] claims:
  - In many systems, computations *are* extremely short
  - There are many sound techniques for decomposing long-running computations in practice
  - Increased micro-controller performance is reducing this problem
  - Where increased micro-controller performance is still not good enough, add more processors

# Time-triggered scheduler: use in practice

```
int main() {
    schInit();           /* configure scheduler — 1 ms tick */

    schAddTask(f, 0, 5); /* prepare to run task f every 5 ms */
    schAddTask(g, 1, 10); /* prepare to run task g every 10 ms */
    schAddTask(h, 3, 15); /* prepare to run task h every 15 ms */

    schStart();          /* start ticking ... */

    while (true) {
        schDispatch();    /* if a task is ready to run, run it */
    }
}
```

# Time-triggered scheduler: implementation

```
#ifndef __SCHEDULER_H
#define __SCHEDULER_H

#include <stdint.h>
#include <bsp.h>
#include <ttSchedConfig.h>

/* Task Control Block structure */
typedef struct schTCB {
    pVoidFunc_t task;
    uint32_t delay;
    uint32_t period;
    uint8_t invocations;
} schTCB_t;

void schInit(void);           // initialise the scheduler
void schStart(void);         // start ticking
void schUpdate(void);        // update after a tick — ISR
void schDispatch(void);      // run the next task
void schAddTask(
    pVoidFunc_t,             // the task to add
    uint32_t,                // the delay in ms
    uint32_t);               // the period
void schRemoveTask(
    uint8_t);                // remove a set from the task set
                             // identifier of the task to remove
void schSleep(void);         // go to sleep to save power

#endif
```

# Acknowledgements

- Pont, M., [Patterns for Time-triggered embedded systems](#), TTE Systems, 2010
- Liu, J., Real-time systems, Prentice Hall, 2000