Microprocessor – Microcontroller
W05.2 – A co-operative scheduler

Nguyen Tran Huu Nguyen

D: Computer Engineering

E: nthnguyen@hcmut.edu.vn

What is a co-operative scheduler?

- A **co-operative scheduler** provides a **single-tasking system** architecture

Operation:
- Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)
- When a task is scheduled to run it is added to the waiting list
- When the CPU is free, the next waiting task (if any) is executed
- The task runs to completion, then returns control to the scheduler

Implementation:
- The scheduler is simple and can be implemented in a small amount of code
- The scheduler must allocate memory for only a single task at a time
- The scheduler will generally be written entirely in a high-level language (such as 'C')
- The scheduler isn't a separate application; it becomes part of the developer's code

Performance:
- Obtaining rapid responses to external events requires care at the design stage

Reliability and safety:
- Co-operate scheduling is simple, predictable, reliable and safe

Introduction

- In this lecture, we discuss technique for creating a co-operative scheduler suitable for use in single processor environments.

- These provide a very flexible and predictable software platform for a wide range of embedded applications, from the simplest consumer gadget up to and including aircraft control systems.

Introduction

Context

- You are developing an embedded application using MCUs.

- The application is to have a time-triggered architecture, constructed using a scheduler.

Problem

- How do you create and use a co-operative scheduler?

Background - Function pointer

- Just as we can determine the starting address of an array of data in memory ➔ we can also find the address in memory at which the executable code for a particular function begins.

- This address can be used as a 'pointer' to the function; most importantly, **it can be used to call the function.**

Background - Function pointer

- Used with care, function pointers can make it easier to design and implement complex programs.
- For example, suppose we are developing a large, safety-critical, application, controlling an industrial plant.
  - If we detect a critical situation, we may wish to shut down the system as rapidly as possible.
  - However, the appropriate way to shut down the system will vary, depending on the system state.
  - What we can do is create a number of different recovery functions and a function pointer. Every time the system state changes, we can alter the function pointer so that it is always pointing to the most appropriate recovery function.
  - In this way, we know that – if there is ever an emergency situation – we can rapidly call the most appropriate function, by means of the function pointer.

Function pointer - Example

```c
void Square_Number(int, int*);
int main(void) {
        int a = 2, b = 3;
        // pFn is a pointer with int and int pointer parameters (return void)
        void (* pFn)(int, int*);
        int Result_a, Result_b;
        pFn = Square_Number; // pFn holds address of Square_Number
        printf("Function code starts at address: %u\n", (tWord) pFn);
        printf("Data item a starts at address: %u\n\n", (tWord) &a);
        // Call 'Square_Number' in the conventional way
        Square_Number(a, &Result_a);
        // Call 'Square_Number' using function pointer
        (*pFn)(b, &Result_b);
        printf("%d squared is %d (using normal fn call)\n", a, Result_a);
        printf("%d squared is %d (using fn pointer)\n", b, Result_b);
        while(1);
}
```

Function pointer - Example

```c
void (*TMR0_InterruptHandler)(void);

void TMR0_SetInterruptHandler(void (* InterruptHandler)(void)){
    TMR0_InterruptHandler = InterruptHandler;
}

void TMR0_Initialize(void){
    TMR0_SetInterruptHandler(TMR0_DefaultInterruptHandler);
}

void TMR0_DefaultInterruptHandler(void){
    // add your TMR0 interrupt custom code
    // or set custom function using TMR0_SetInterruptHandler()
}

void TMR0_ISR(void) {
    if(TMR0_InterruptHandler) {
        TMR0_InterruptHandler();
    }
}
```

Solution

A scheduler has the following key components:

➢ The scheduler data structure.

➢ An initialization function.

➢ A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.

➢ A function for adding tasks to the scheduler.

➢ A dispatcher function that causes tasks to be executed when they are due to run.

➢ A function for removing tasks from the scheduler (not required in all applications).

We consider each of the required components in this section.

Overview

- Before discussing the scheduler components, we consider how the scheduler will typically appear to the user.

- To do this we will use a simple example
  - A scheduler is used to flash a single LED on and off repeatedly: on for one second off for one second etc.

The key components from a simple scheduled application

```c
void main(void) {
        // Set up the scheduler
        SCH_Init_T2();
        // Prepare for the 'Flash_LED' task
        LED_Flash_Init();
        // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
        // - timings are in ticks (1 ms tick interval)
        // (Max interval / delay is 65535 ticks)
        SCH_Add_Task(LED_Flash_Update, 0, 1000);
        // Start the scheduler
        SCH_Start();
        while(1)  {
                SCH_Dispatch_Tasks();
        }
}
void SCH_Update(void)  {    }
```

1. The LED is switched on and off by means of a 'task' LED_Flash_Update(). Thus, if the LED is initially off and we call LED_Flash_Update() twice, we assume that the LED will be switched on and then switched off again.

2. To obtain the required flash rate, the scheduler calls LED_Flash_Update() every second *ad infinitum*.

3. We prepare the scheduler using the function SCH_Init_T2().

4. After preparing the scheduler, we add the function LED_Flash_Update() to the scheduler task list using the SCH_Add_Task() function. At the same time we specify that the LED will be turned on and off at the required rate

5. The timing of the LED_Flash_Update() function will be controlled by the function SCH_Update(), an interrupt service routine triggered by the overflow of Timer 2.

6. The 'Update' ISR does not execute the task: it calculates when a task is due to run and sets a flag. The job of executing LED_Flash_Update() falls to the dispatcher function (SCH_Dispatch_Tasks()), which runs in the main ('super') loop

The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task.

```c
typedef data struct {
// Pointer to the task (must be a 'void (void)' function
        void ( * pTask)(void);
// Delay (ticks) until the function will (next) be run
        tWord Delay;
// Interval (ticks) between subsequent runs.
        tWord Period;
// Incremented (by scheduler) when task is due to execute
        tByte RunMe;
} sTask;
#define SCH_MAX_TASKS (1)
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

The size of the task array

- You **must** ensure that the task array is sufficiently large to store the tasks required in your application, by adjusting the value of SCH_MAX_TASKS.

- For example, if you schedule three tasks as follows:
  - SCH_Add_Task(Function_A, 0, 2);
  - SCH_Add_Task(Function_B, 1, 10);
  - SCH_Add_Task(Function_C, 3, 15);

then SCH_MAX_TASKS must have a value of three (or more) for correct operation of the scheduler.

- Note also that, if this condition is not satisfied, the scheduler will generate an error code.

The initialization function

- Like most of the tasks we wish to schedule, the scheduler itself requires an initialization function.

- This performs various important operations – such as preparing the scheduler array and the error code variable.

- The main purpose of this function is to set up a timer that will be used to generate the regular 'ticks' that will drive the scheduler.

The 'Update' function

- The 'Update' function is the scheduler ISR. It is invoked by the overflow of the timer.

- Like most of the scheduler, the update function is not complex.

- When it determines that a task is due to run, the update function increments the RunMe field for this task: the task will then be executed by the dispatcher.
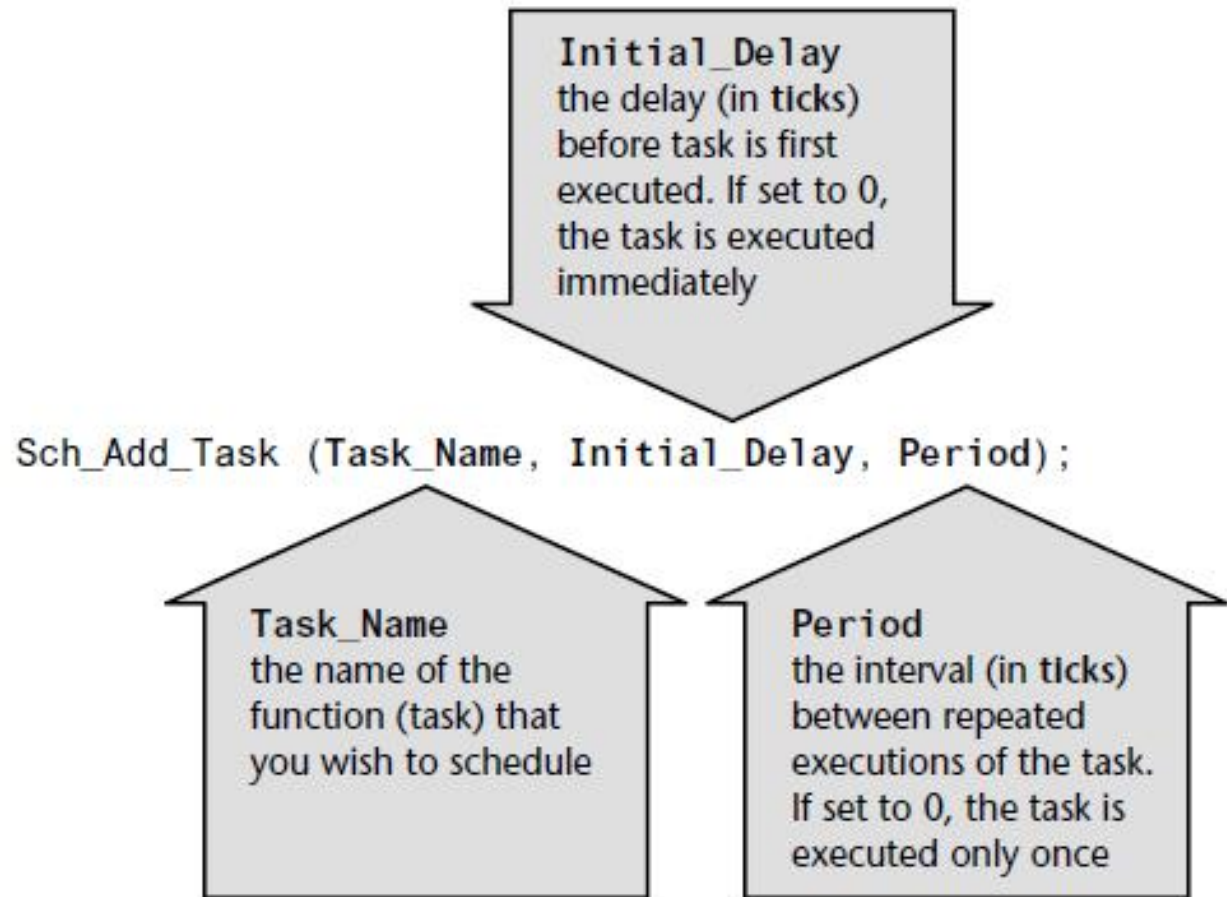
A scheduler 'update' function

```c
void SCH_Update(void)  {
  tByte Index;
  // NOTE: calculations are in *TICKS* (not milliseconds)
  for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
  // Check if there is a task at this location
    if (SCH_tasks_G[Index].pTask) {
        if (SCH_tasks_G[Index].Delay == 0) {
        // The task is due to run
                SCH_tasks_G[Index].RunMe += 1; // Inc. the 'RunMe' flag
                if (SCH_tasks_G[Index].Period) {
                        // Schedule periodic tasks to run again
                        SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
        } else {

                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
        }
      }
    }
}
```

The 'Add Task' function

The 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s).

Initial_Delay
the delay (in ticks) before task is first executed. If set to 0, the task is executed immediately

Sch_Add_Task (Task_Name, Initial_Delay, Period);

Task_Name
the name of the function (task) that you wish to schedule

Period
the interval (in ticks) between repeated executions of the task. If set to 0, the task is executed only once

Some examples

- **SCH_Add_Task(Do_X, 1000, 0);**
  - This set of parameters causes the function Do_X() to be executed once after 1,000 scheduler ticks

- **Task_ID = SCH_Add_Task(Do_X, 1000, 0);**
  - This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see SCH_Delete_Task() for further information about the removal of tasks from the task array)

- **SCH_Add_Task(Do_X, 0, 1000);**
  - This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; the task will first be executed as soon as the scheduling is started

- **SCH_Add_Task(Do_X, 300, 1000);**
  - This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; task will be first executed at T = 300 ticks, then 1,300, 2,300 etc

An implementation of the scheduler 'add task' function

```
tByte SCH_Add_Task(void (code * pFunction)(), const tWord DELAY, const tWord PERIOD) {
   tByte Index = 0;
   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS)) {
      Index++;
   }
   if (Index == SCH_MAX_TASKS)  {// Have we reached the end of the list?
      // Task list is full, Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
      return SCH_MAX_TASKS; // Also return an error code
   }
   // If we're here, there is a space in the task array
   SCH_tasks_G[Index].pTask = pFunction;
   SCH_tasks_G[Index].Delay = DELAY;
   SCH_tasks_G[Index].Period = PERIOD;
   SCH_tasks_G[Index].RunMe = 0;
   return Index; // return position of task (to allow later deletion)
}
```

The 'Dispatcher'

- The 'Update' function does not execute any tasks.

- The tasks that are due to run are invoked through the 'Dispatcher' function.

An implementation of the scheduler 'dispatch task' function

```
void SCH_Dispatch_Tasks(void) {
  tByte Index;
  // Dispatches (runs) the next task (if one is ready)
  for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
    if (SCH_tasks_G[Index].RunMe > 0) {
      (*SCH_tasks_G[Index].pTask)(); // Run the task
      SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag
      // Periodic tasks will automatically run again
      // - if this is a 'one shot' task, remove it from the array
      if (SCH_tasks_G[Index].Period == 0) SCH_Delete_Task(Index);
    }
  }
  SCH_Report_Status(); // Report system status
  SCH_Go_To_Sleep(); // The scheduler enters idle mode at this point
}
```

Do we need a Dispatch function?

- The use of both the 'Update' and 'Dispatch' functions may seem a rather complicated way of running the tasks. Specifically, it may appear that the Dispatch function in unnecessary and that the Update function could invoke the tasks directly.

- However, the split between the Update and Dispatch operations is necessary, **to maximize the reliability of the scheduler in the presence of long tasks.**

- Suppose we have a scheduler with a tick interval of 1 ms and, for whatever reason, a scheduled task sometimes has a duration of 3 ms.

  – If the Update function runs the functions directly then – all the time the long task is being executed – the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected and may mean that two (or more) tasks are not scheduled to execute at all.

  – If the Update and Dispatch function are separated, system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

The 'Delete Task' function

- When tasks are added to the task array, SCH_Add_Task() returns the position in the task array at which the task has been added:
  - Task_ID = SCH_Add_Task(Do_X,1000,0);
- Sometimes it can be necessary to delete tasks from the array. To do so, SCH_Delete_Task() can be used as follows:
  - SCH_Delete_Task(Task_ID);

An implementation of the scheduler 'delete task' function

```
bit SCH_Delete_Task(const tByte TASK_INDEX) {
    bit Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0) {
        // No task at this location...
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;
        // ...also return an error code
        Return_code = RETURN_ERROR;
    } else {
        Return_code = RETURN_NORMAL;
    }
    SCH_tasks_G[TASK_INDEX].pTask = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay = 0;
    SCH_tasks_G[TASK_INDEX].Period = 0;
    SCH_tasks_G[TASK_INDEX].RunMe = 0;
    return Return_code; // return status
}
```

Reporting errors

- Hardware fails; software is never perfect; errors are a fact of life.
- To report errors at any part of the scheduled application, we use an (8-bit) error code variable **Error_code_G**, which is defined as follows:
  - tByte Error_code_G = 0;
- To record an error we include lines such as:
  - Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
  - Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
  - Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
  - Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
  - Error_code_G = ERROR_SCH_LOST_SLAVE;
  - Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
  - Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;

An implementation of the scheduler 'report status' function

```c
void SCH_Report_Status(void) {
#ifdef SCH_REPORT_ERRORS // ONLY APPLIES IF WE ARE REPORTING ERRORS
// Check for a new error code
if (Error_code_G != Last_error_code_G) {
   // Negative logic on LEDs assumed
   Error_port = 255 - Error_code_G;
   Last_error_code_G = Error_code_G;
   if (Error_code_G != 0) {
      Error_tick_count_G = 60000;
   } else {
      Error_tick_count_G = 0;
   }
} else {
   if (Error_tick_count_G != 0) {
      if (--Error_tick_count_G == 0) {
         Error_code_G = 0; // Reset error code
      }
   }
}
#endif
```

What does that error code mean?

- The forms of error reporting discussed here are low-level in nature and are primarily intended to assist the developer of the application or a qualified service engineer performing system maintenance.

- An additional user interface may also be required in your application to notify the user of errors, in a more user-friendly manner.

Reliability and safety implications

- Make sure the task array is large enough

- Take care with function pointers

- Dealing with task overlap
  - Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second and Task B every three seconds. We assume also that each task has a duration of around 0.5 ms.
  - Suppose we schedule the tasks as follows (assuming a 1ms tick interval):
    - SCH_Add_Task(TaskA,0,1000);
    - SCH_Add_Task(TaskB,0,3000);

Reliability and safety implications

- Dealing with task overlap
  - In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A. This will mean that if Task A varies in duration, then Task B will suffer from 'jitter': it will not be called at the correct time when the tasks overlap.
  - Alternatively, suppose we schedule the tasks as follows:
    - SCH_Add_Task(TaskA,0,1000);
    - SCH_Add_Task(TaskB,5,3000);
  - Now, both tasks still run every 1,000 ms and 3,000 ms (respectively), as required. However, Task A is explicitly scheduled always to run 5 ms before Task B. As a result, Task B will always run on time.

Guidelines for predictable and reliable scheduling

1 For precise scheduling, the scheduler tick interval should be set to match the 'greatest common factor' of all the task intervals.

2 All tasks should have a duration less than the schedule tick interval, to ensure that the dispatcher is always free to call any task that is due to execute. Software simulation can often be used to measure the task duration.

3 In order to meet Condition 2, all tasks **must** 'timeout' so that they cannot block the scheduler under any circumstances. Note that this condition can often be met by incorporating, where necessary, a **LOOP TIMEOUT** or a **HARDWARE TIMEOUT** in scheduled tasks.

Please remember that this condition also applies to any functions called from within a scheduled task, including any library code provided by your compiler manufacturer. In many cases, standard functions (like printf()) do not include timeout features. They must not be used in situations where predictability is required.

4 The total time required to execute all the scheduled tasks must be less than the available processor time. Of course, the total processor time must include both this 'task time' and the 'scheduler time' required to execute the scheduler update and dispatcher operations.

5 Tasks should be scheduled so that they are never required to execute simultaneously: that is, task overlaps should be minimized. Note that where **all** tasks are of a duration much less than the scheduler tick interval, and that some task jitter can be tolerated, this problem may not be significant.

Overall strengths and weaknesses

- The scheduler is simple and can be implemented in a small amount of code.

- The applications based on the scheduler are inherently predictable, safe and reliable.

- The scheduler is written entirely in 'C': it is not a separate application but becomes part of the developer's code.

- The scheduler supports team working, since individual tasks can often be developed largely independently and then assembled into the final system.

- Obtain rapid responses to external events requires care at the design stage.

- The tasks cannot safely use interrupts: the only interrupt that should be active in the application is the timer-related interrupt that drives the scheduler itself.