**DC Fan/Motor Control Using Bluetooth Based Application (Android)**

Author

Prasad Khedekar (G00973357)

**Acknowledgements**

I am thankful to my Professor Dr. Peter Pachowicz for all his support and kind guidance. I am also grateful to him for clearing my basic concepts related to the subject and project, specifically in terms of design of the scheduler, his timely advice and suggestions which led to the successful working of this project.

**Abstract**

The scheduler is the most important component of the kernel of any real-time embedded system. Its responsibility is to ensure that the computing resources are kept busy by scheduling tasks whenever they become ready. The underlying scheduling algorithm makes sure that tasks are scheduled in such a way that they meet their specific deadlines and thereby maintain system integrity.

In this report, I have explained the design process of implementing a scheduler for tasks involved in the application of controlling a DC Fan/Motor using Bluetooth based Android application.

**Problem Statement/Description of System Functionality:**

* The aim of the project is to design a scheduler for tasks involved in controlling a DC Fan using an Android-based application over Bluetooth through a smartphone.
* In this prototype, it is possible to remotely control the speed of the fan over Bluetooth from level 0 (lowest speed) through level 5 (highest speed), calculate the current speed in RPM and display status of current level on the Android application.
* Furthermore, it also features an emergency mode wherein the data from a smoke sensor is sampled to gauge fire risks and implement safe shutdown of the fan along with an alert on the user’s connected smartphone.
* This application gives rise to a bunch of tasks with inter-dependencies which are triggered at certain specific events. This is the idea/motivation behind implementing and experimenting with a scheduler to manage these tasks.

**Scheduler Approach:**

* Since the application works on the triggering of events for tasks to be scheduled, the scheduler is modelled as a priority-preemptive event-driven scheduler.
* It features 3 tasks which are scheduled on the basis of certain events occurring and their priorities once they are activated.
* The system uses event flags to signal occurrence of events, which then act as signals for initiating/executing a particular task.
* The scheduler is invoked periodically based on timer intervals. It then decides which task to run and if necessary, preempts the current running task by performing a context switch and then scheduling the chosen task.
* The scheduling algorithm is based on event flags and priorities of tasks.
* The tasks utilize shared global variables for data exchange and synchronization.
* Interrupts are the primary source for flagging of events, which act as required conditions for the initiation/execution of a task.
* The key is to keep ISRs short and atomic such that the impact on the execution state of the system is minimal.
* This is achieved by splitting the ISRs into the actual flagging of event (done inside the ISR) and a worker task of lower priority, which then performs the operations on data and outputs the results.
* The ISRs pass data on to the 3 worker tasks by accessing the shared global variables and modifying them.
* To avoid race conditions, the system does not support nesting of interrupts.

**Scheduler Architecture:**



**Scheduler Architecture – Tasks:**

Task0: This task is used to monitor the sensor data. The ADC interrupt sets the flag for this task if the threshold is crossed, following which it initiates the emergency shutdown of the fan and also initiates the sending of alert message.

Task1: This task is used to change the speed of the DC Fan based on the required level. It modifies the duty cycle value of the outgoing PWM signal to the fan based on the preset values associated with each level. It then sets the flag for Task2 to be initiated/executed next.

Task2: This task is used to transmit the information from the MCU to the connected smartphone over Bluetooth using UART communication. The transmitted information consists of the current level at which the fan is rotating, which is sent as a text and displayed on the user’s device.

**Scheduler Architecture – ISRs:**

Watchdog Timer Interval ISR: In this application, the WDT is used to provide the periodic timer ticks for invoking the scheduler. The reason for using the WDT in Timer interval mode is its higher priority interrupt than other Timer module interrupts within the MSP430. The scheduling algorithm is implemented inside the ISR, which then initiates the appropriate task to be scheduled based on the status of the event flags. Additionally, it performs the required context switching for the tasks.

USCIA0 (UART) ISR: This ISR is used to serve the received information interrupt from the connected device over Bluetooth using UART. It sets the level as per the received request, then sets the flag for Task1.

ADC12 ISR: This ISR is used to service the interrupt raised by the in-built ADC module when it finishes a conversion and loads the result. The result is compared with the threshold, and a flag for Task0 is raised, if it is found to be above the threshold.

Port2 ISR: This is a software ISR for measuring the Tachometer pulses from the fan. It counts the number of pulses by counting the number of rising/falling edges on the input line. This is then used for calculating the Fan RPM.

Timer0 A0 ISR: The Timer0 module is programmed to go off every 100ms, after which it is serviced using this ISR. It is basically used to calculate the current RPM of the Fan based on the count of the number of Tachometer pulses from the Fan. The Fan RPM is then updated and the counter set to zero.

**RTOS Concepts & Objects Used:**

* Binary semaphores in terms of event flags are used to synchronize tasks and their activities.
* A particular task is initiated only if its corresponding semaphore is signaled, after which it executes and then releases (clears) the flag for next instantiation.
* The tasks make use of shared global variables for data exchange and message passing.
* The ISRs access these global variables on occurrence of interrupts. To achieve system integrity, the ISRs are short and atomic in their operations on global variables and are split into worker tasks.

**Task Switching/Context Switching:**

* An important function of the scheduler is to switch tasks as per the scheduling algorithm which involves the concept of context switching.
* On a task switch, the registers, PC, Status Register, Stack Pointer form the context of a task which needs to be saved so that when it is resumed, it can continue its execution from the point of switch.
* In this application, Task0 has the highest priority, followed by Task1 and then Task2. All the tasks are modelled as infinite loops waiting on their corresponding flag to occur.
* The scheduling algorithm checks the status of the flags, then initiates the execution of the corresponding task and performs the required context switch, if necessary. Task0 is selected to execute as a background task if no other task flag is raised.

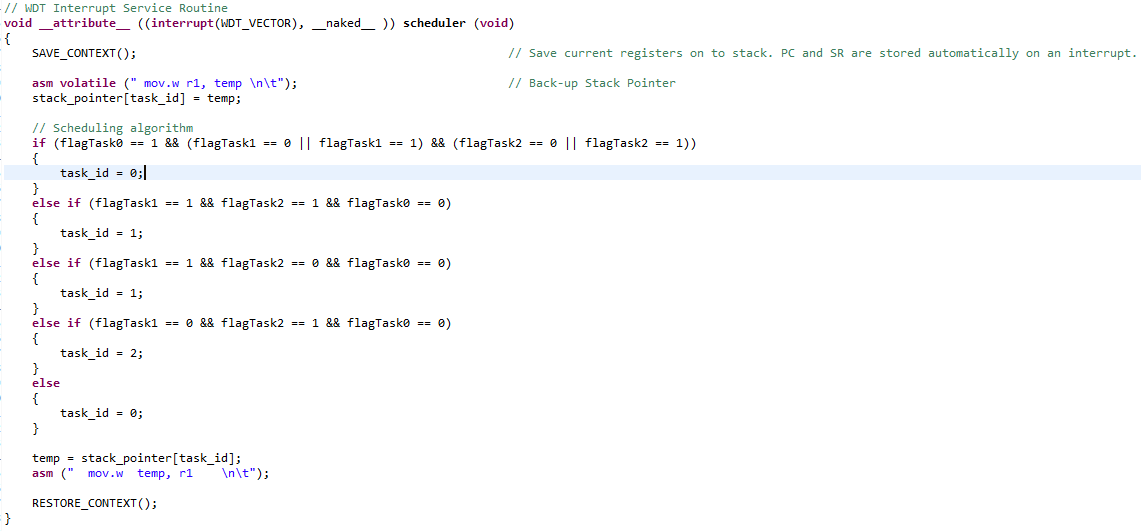
**Task Switching/Context Switching – Mechanism:**



* MSP430F5529 has a total of 8KB of RAM. It is divided among the tasks and the scheduler as shown above.
* Whenever a task is switched, the scheduler saves its context in its allocated block of RAM.
* It then proceeds to access the allocated RAM of the chosen task to be run, to restore its context, following which it correctly resumes its execution.
* Stack Pointer (SP) is used to access the correct stack location at which a task has its context saved.
* During initialization of each of the tasks, they are allocated specific space in the RAM at which they need to store: PC (r0), SR (r2), and general purpose registers (r4-r15). At initialization, the PC points to the task function address, from which it should begin execution.
* Task0 is then initiated as the first task to be executed in the main(). From there on, at every WDT tick, the scheduler is invoked and the next task is run, if its flag is raised.

When the scheduler is invoked, it takes the following steps:

* It saves the context of the current running task i.e. general purpose registers (r4-r15) on its dedicated stack. Note that we do not need to explicitly save PC and SR as they are automatically saved on the stack when the interrupt occurs.
* It then saves the SP, which points to the Top of the stack, in a variable so that it knows where to point to in the stack when this task is resumed.
* The scheduling algorithm is run to determine which task is to be scheduled next.
* Once the next task is decided, the SP address, which was previously stored in that variable for that task when it was switched out, is restored in the SP register (r1) so that it points to the saved context of the next task.
* Now that it knows where the saved context lies on the stack, it simply restores (pops) the content for that task so that it can resume its execution.



Code Snippet showing Context Switching

**MCU Information:**

* The MCU used in this application is MSP430F5529. Its features are: 16-Bit Ultra-Low-Power Microcontroller, 128KB Flash, 8KB RAM, USB, 12Bit ADC, 2 USCIs
* The reason for choosing this MCU was its ample amount of available FLASH and RAM memory, which is utilized for the code and the dedicated RAM allocation to tasks.

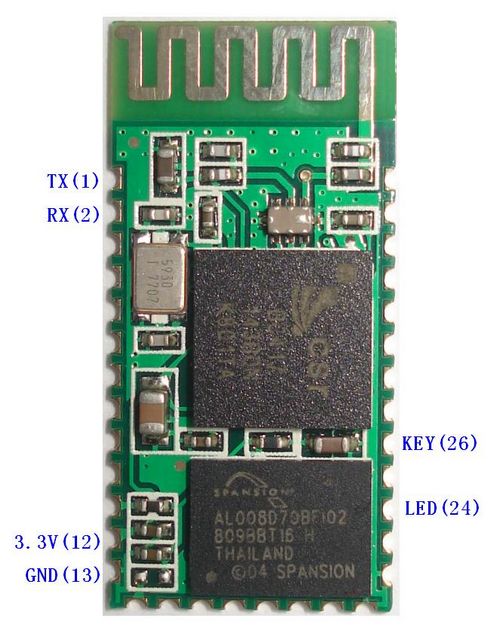
The following MCU resources have been used in this project:

* WDT module in Timer Interval mode
* UART module for Bluetooth communication
* ADC module for converting the analog sensor output to digital input for MCU
* Timer1 module for PWM signal generation
* Timer0 module to provide 100ms intervals used for refreshing the RPM values
* Ports 1 and 2, where port 2 receives the Tachometer signal from the Fan

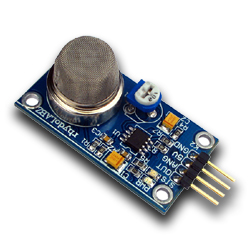
**Experimental Setup:**

Components used apart from the MCU:

* Bluetooth Transceiver HC-06

Serial port Bluetooth, Drop-in replacement for wired serial connections, transparent usage. Simple usage for a serial port replacement to establish connection between MCU and GPS/Mobile phones, PC to embedded projects and etc.

Key Features

* Bluetooth protocol:  Bluetooth Specification v2.0+EDR
* Frequency:  2.4GHz ISM band
* Smoke/Gas Sensor MQ-2

This flammable gas and smoke sensor detects the concentrations of combustible gas in the air and outputs its reading as an analog voltage. The sensor can measure concentrations of flammable gas of 300 to 10,000 ppm. The sensor can operate at temperatures from -20 to 50°C and consumes less than 150 mA at 5 V.

* Cooler Master DC fan with specifications as: DC 12V, 0.30 A
* Battery (9V)
* Android application for controlling the speed of the fan

**Experimental Setup – Block Diagram:**



* The experimental setup largely involved testing the task switching of the underlying tasks based on the commands (interrupts) generated from the user’s connected smartphone.
* The fan’s speed was adjusted as per the level required by the user.
* On the occurrence of an emergency, task switching was observed from Task0 to Task1 to lower the speed of the fan, and then finally onto Task2 where the emergency signal is sent out to the user’s connected smartphone.
* The handling of events, ISRs, tasks by the scheduler was tested based on various different test scenarios.

**Results:**

* When the system is idle, it runs Task0 by default as the background task. On receiving interrupts in the form of commands from the smartphone device, the application successfully switches task to run Task1, which handles changing the speed of the fan.
* Following the completion of the speed change task, Task2 is observed to flag off, which handles the feedback to user’s device in the form of status of the fan displayed on the screen.
* The system also promptly responds to situation where the smoke sensor senses a higher quantity of smoke, by immediately lowering the fan speed to its lowest level and sending out an alert signal to the user’s device about the emergency. The tasks were observed to successfully switch to achieve this state.

**Issues Faced & Resolutions:**

Issue: A peculiar issue faced where the WDT fires during the CSTARTUP code even before entering the main()

Resolution: The WDT had to be turned off during the system pre-initialization to avoid system reset during the initialization/startup C code.

Issue: Maintaining stack integrity during ISR calls. The ISR automatically performs context switch including certain registers, which led to stack overflowing out of its dedicated space.

Resolution: The automatic context switch had to be turned off using the ‘naked’ attribute in GCC. It then stores just the PC and SR.

Issue: The monitor keyword can be used to declare functions that cannot be interrupted, which provides a good implementation of semaphores for protecting the shared global variables while they are being accessed by the tasks. Unfortunately, the TI-GCC compiler issued errors on its usage.

Resolution: Made use of task flags and appropriate conditions to ensure contention-free access.

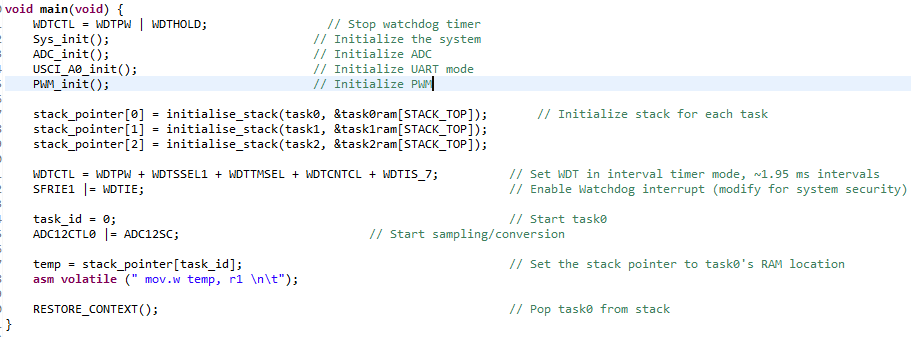
**Conclusions:**

* The biggest challenge proved to be implementing the event-driven scheduler and handling the ISRs to achieve task switching behavior. The introduction of the concepts of event flags and splitting of ISRs into worker tasks to make them atomic proved to be beneficial in realizing the scheduler.
* Further enhancements could include implementation of message queues protected by dedicated semaphores as the number of tasks increases and the amount of shared data, messages increases to make the system scalable.
* As the number of tasks increases, a global protected event register can be used to store event flags. The scheduler would then make scheduling decisions based on this event register.

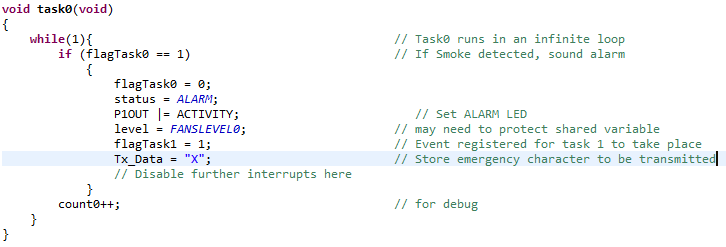
**Code Snippets:**



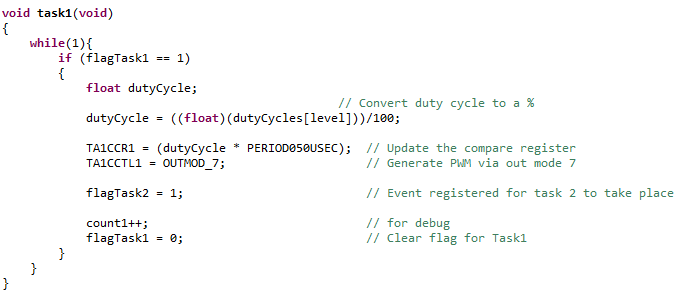
Macros for context switching



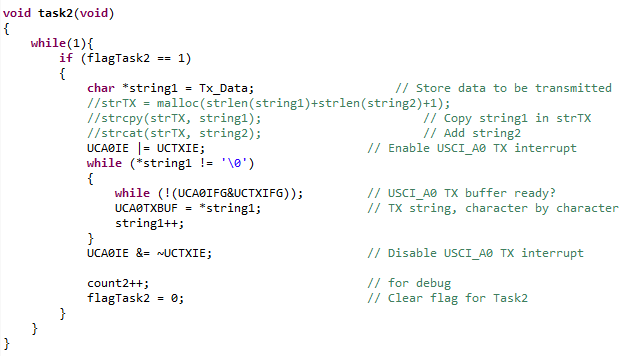
Main() initializations



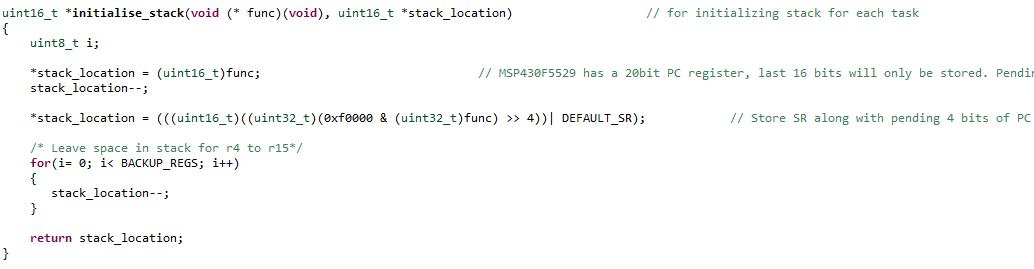
Task0



Task1



Task2



Stack Initialization