RTOS Implementation for OTA Updates

Project Subsystems:

1. ESP32 (as Internet Node)
2. STM Bootloader (Bare Metal - RTOS Unaware)
3. STM Application Code (RTOS Aware)

Board Used:

1. STM32F407VGT6/STM32F411VE
2. ESP32

RTOS used: CMSIS RTOS V2

RTOS Elements:

1. Message Queue
2. Event Flags
3. Mutex

Need for RTOS: To provide Multi-Tasking and efficient resource management (like for UART) to the system. It also provides synchronization between the tasks which are based on the producer-consumer problem. This is mainly solved using Message Queue. Events or Notifications are also required for unblocking the particular task by ISRs.

This document mainly deals with the third subsystem i.e. STM Application Code (RTOS Aware). For the deployment of the RTOS Aware Application in the STM32F4 Board, CMSIS RTOS Version 2 (latest) is used.

Tasks:

There are three tasks

1. **"*accelTask*"**

   Priority : *osPriorityNormal*
   Stack Size : 960
   Method : AccelReadThread
   Type : Time triggered
   Periodicity : Periodic (5 sec)

   Description: This task is a Producer Task which is used to read the values (x,y,z) of accelerometer via SPI or I2C (depending upon the board). After reading the values, each time, it pushes the readings to a Message Queue via "*osMessageQueuePut*". If message Queue is full, this task will be blocked till the Consumer Task (***sendTask***) does not read the values from the message queue.
   This provides a synchronization mechanism between producer and the consumer task. Thus consumer task will always be scheduled after the producer task which will be scheduled in every 5 seconds.
   If the producer task fails to put the value on the queue, it will indicate the user about the error by turning ON the Red LED.

2. "***sendTask***"

Priority	**:**	*osPriorityNormal*
Stack Size	**:**	240
Method	**:**	TransmitUartThread
Type	**:**	Event triggered
Periodicity	**:**	Aperiodic

Description: This task is a Consumer Task which is used to get the accelerometer values from the message queue using "*osMessageQueueGet*", and transmit those values to the ESP32(Internet Node) over UART. Initially, when the queue is empty, this task will be blocked (as it will try to read from the empty queue). It will be unblocked whenever producer (***accelTask***) produces the message. Thus providing complete synchronization with the producer task and making the consumer task to be scheduled only after the producer task.
If the consumer task fails to get the values from the queue, it will indicate the user about the error by turning ON the Red LED.
It must be noted that, this task is using UART which is a shared resource among the Tasks, thus making it a critical resource. So to avoid *race condition* and to provide *mutual exclusion* for the critical resource (UART) Mutex is used. Similarly, if there is any error while using Mutex, it will be indicated by turning ON the red LED.

3. "***updateTask***"

Priority	**:**	*osPriorityHigh*
Stack Size	**:**	480
Method	**:**	UpdateHandleThread
Type	**:**	Event triggered
Periodicity	**:**	Aperiodic (Sporadic)

Description: This task is an event based, sporadic task which will be unblocked and scheduled only when user generates an external interrupt by pressing the pushbutton. Initially, this task will be blocked for the Event Flag.
It is a high priority task so when unblocked/scheduled, no other task will run and thus it can surely meet its deadline (*Sporadic*).
It is a **multi-state** task which has two different states or jobs to perform according to the event flag received.
The two states are:

a. Check for Update: Whenever user presses the pushbutton for the first time, Interrupt will be generated and the ISR will wake /unblock this task by Event Flag *0x00000001*. This state mainly deals with the sending of the *"check_update"* request (over UART) to the ESP which will further check with the server whether any new update is available or not.
After sending the request "*check_update*", this task will wait for response from ESP (which in turn will get response from server) for ACK or NACK. Server will send Authentication

Frame as ACK and if anything else is received, it will be treated as NACK. Authentication Frame is explained below in the text.

ESP will send whatever received from the server to the STM. Authentication of this response is also done by this task using a user defined function "*parsing_update_request*". Here, if the task finds the response to be authentic, it will update the "*Signature Structure*" stored in the last sector of the flash. The "*sector_no*" field and the "sector_addr" field of the structure is updated to the sector number and the sector address of the flash where the updated firmware will be flashed by the bootloader code during the update (subsystem 2). Similarly, the version of the updated firmware is stored in the "*version*" field of the structure. At last the "*updateAvailable*" field of the structure is made "*TRUE*" which will force the bootloader code to update the current firmware either in the next boot or whenever requested by the user (section b). For telling the ESP that the update is legitimate, this task will send the reverse of the "secure_id" (specific to the board), which is received in the authentication frame. At last, to indicate the user that a new authentic update is available, this task turns ON the Green LED present in the board. Finally, it will be blocked again for the next Event Flag which will be generated when the user wants to update the new authentic firmware and again presses the pushbutton.

If the task finds the response from the ESP to be spurious or inauthentic, it will just send the "secure_id" as it is (as NACK). Thus, this task will again be blocked for the Event Flag generated when user checks for the update next time.

Also, as this task is again using UART for the communication with the ESP, so Mutex is required for the mutual exclusion or atomicity of critical resource (UART). Red LED will be turned ON in case of any error while acquiring or releasing the Mutex.

b. <u>Start Update (Jump to Bootloader)</u>: The task will be unblocked in this state only by the Event Flag *0x00000002*, when user presses the pushbutton and a new authentic firmware update is available. It means whenever the pushbutton is pressed, ISR of the external interrupt generated will first check the update available flag in the signature structure, and then send the event flag to unblock this task in this state, only if the flag is true. If it is false, above (section a) state will be unblocked.

   Thus, this state will get the control only when the Green LED is ON and user presses the pushbutton to download and update the newly available firmware. Its job is just to jump the control to the bootloader (present in the sector 0 of the flash) so that update can be written in the flash.

RTOS Aware ISR:

For this system only one interrupt is configured which will be generated as an External Line interrupt by the pushbutton. It will be used to unblock the "*updateTask*" in different states by sending the Event Flags. This ISR will first check the status of the "*updateAvailable"* flag by reading the signature structure from the last sector of the flash. If the flag is true i.e. new update is available, it will unblock the task in the second state (Start Update) by sending the flag mask 0x00000002U. If it is false, task will be unblocked in the second state (Check for Update) i.e. user wants to check for new updates, by sending the flag mask 0x00000001U. It can be inferred that this ISR is contributing to the **Top-Half** of the interrupt processing and the task "*updateTask*" to the **Bottom Half** of the interrupt processing. As it is an ISR, so any Blocking Call is not used here. This ISR is RTOS Aware as various RTOS services are used here like Event Flags. Also it is not called directly from the interrupt Handler but it is a callback called from the internal layer of the RTOS whenever any interrupt occurs.

RTOS Services/ Elements Used:

I. Event Flags:

Mask Used	**:**	0x00000003U (Flag bit 1 and 2 enabled)
Flag 0x00000001	**:**	**"**Check for Update" state of **"*accelTask*"**
Flag 0x00000002	**:**	**"**Start Update" state of **"*accelTask*"**

These are similar to the Notifications in **FreeRTOS** or Signals in the **Linux** System. Here, it is used to unblock the task and notify it about certain events (interrupts) so that the event based task can work.

II. Message Queues:

Size	**:**	4 blocks, each of a 32 bytes
Producer Task	**:**	**"*accelTask*"** for reading Accelerometer Data
Consumer Task	**:**	**"*sendTask*"** for sending data to UART

Here, message queue is mainly used to provide **the Task-To-Task** synchronization and also to give a basic FIFO communication model. One task (producer) sends the data in the message queue and another task (consumer) gets the data from it. As a producer cannot give/put data in a full message queue and consumer cannot take/get data from an empty message queue, so synchronization between the producer and the consumer is achieved. Consumer will always start executing after producer.

III. MutEx:

In this model, UART is a shared resource among various tasks which for exchanging information with the internet node (ESP32). As it is a critical resource and must be accessed atomically, MutEx (Mutual Exclusion) is used. A task can only enter the critical section (UART) if and only if it acquires a Mutex and should also release the Mutex while leaving the critical section. If a task tries to enter the critical section while another task is already present in the critical section (Mutex is already locked), it will try to lock the already locked Mutex, and hence will be blocked in the waitqueue of the Mutex till the Mutex is released/unlocked. Thus it will prevent any

inconsistent data to be exchanged between the nodes and finally avoiding the Race Condition.

Authentication Frame: This data frame is send by the OTA server to the internet node (ESP32) and finally to the target board (STM32F4) for the verification of the available firmware update. Here, '$' (dollar) symbol is used as a token or delimiter between fields of the frame. The basic structure of this frame is:

$**<*BoardName*>**$<*FirmwareVersion*>**$**<*SecureID*>**$**<*FlashSectorNo*>**$**<*FlashSectorAddress*>**$**<*0000..000*>

This is a 64 byte frame which has all the information related to the available update. Each field is described below:

- BoardName: This is the name of the board for which a new update is available. Default name for verification is stored in the current application code running on the board. If the default name and the board name received in the authentication frame does not match, the update is considered as invalid.
  eg. STM32F407VGT6, STM32F411VE, etc

- FirmwareVersion: It specifies the version of the new firmware available on the OTA server. The version of the current running firmware is stored in the Signature Structure stored in the last sector of the flash. The **"*updateTask*"** will read the current firmware version from the flash and compare it to the received version from the server. If both are found same, the firmware is up to date and there is no need to update. Thus authentication frame will be ignored and the update request will be halted.
  eg. 1.0, 1.3, 2.5, etc.

- SecureID: This is a unique ID for every product which is supports this OTA update feature. It helps the server to uniquely identify each and every device it has to serve for the updates. It is also stored in theCrrent running Application and is verified from there. It is also used as ACK/NACK send as a response of the authentication frame to the server.
  eg. A1B2C3D4

- FlashSectorNo: It tells the bootloader to which sector the new update is to be written. It is also stored in the signature structure of the flash. This address changes alternatively with every new update. Suppose, the current running firmware is stored in the sector 2 of the flash, the flash address of the new update will be sector 4 and this update will be flashed in this sector. After updating a firmware, if again any update comes, it will again be stored in the sector 2 defined by the authorization frame. This procedure is followed in every new update.

- FlashSectorAddress: Similar to the FlashSectorNo, there is a field in the Authentication frame named as FlashSectorAddress. It stores the address of the flash where the new updated

firmware is to be stored. It is also stored in the signature structure and changes alternatively with every update like sector number.

- <0000..000>: The standard size of the packet decided for the transmission from ESP to STM is 64 bytes. So, to completely fill the buffer, padding of zeros/nulls/stars is given to the remaining space.

Points To Remember:

- The stack size of each task is to be decided so that there is enough space for the local variables and for further activation records (stack frames). So, a task having more local data, function calls, and processing should be given more stack size. Failing to do so will give the **HardFault**.
- While using any shared resource (here, UART), Mutex should be used to avoid Race Condition and Inconsistency of data.
- Message queues should have enough space to store the sufficient amount of data. As, a full message queue can lead to the loss of data coming from the sensors.
- While writing any data to the flash (Signature Structure or updated firmware), first the flash should be erased (particular sector) then unlocked. After writing is complete, make sure to lock the flash again.
- The above point should also be followed while erasing the flash sector.
- Don’t forget to disable interrupts before writing to the flash and enable them after writing.
- Event Flags should be cleared manually, if they are not getting cleared automatically.