Sistemas Operacionais de Tempo Real

Um sistema operacional em tempo real (RTOS - Real-Time Operating System) é um tipo de sistema operacional projetado para lidar com tarefas que têm requisitos temporais rígidos. Isso significa que ele deve ser capaz de responder a eventos e executar tarefas dentro de prazos predeterminados e garantidos. Os RTOSs são amplamente utilizados em sistemas embarcados, automação industrial, dispositivos médicos, telecomunicações e uma variedade de outras aplicações onde o tempo de resposta é crítico.

O que é um RTOS?

Um RTOS é uma plataforma de software que fornece funcionalidades essenciais para suportar o desenvolvimento de sistemas em tempo real. Ele gerencia os recursos do sistema, como CPU, memória e dispositivos de E/S, de forma a garantir a execução de tarefas dentro de prazos específicos. Os RTOSs são caracterizados por sua capacidade de determinismo, escalabilidade e eficiência em termos de utilização de recursos.

Utilizações do RTOS:

1. **Controle de Processos Industriais**: Em fábricas e plantas industriais, os RTOSs são usados para controlar processos em tempo real, como automação de linhas de produção e sistemas de monitoramento.
2. **Sistemas Embarcados**: Dispositivos como sistemas de controle de automóveis, dispositivos médicos implantáveis, sistemas de navegação e equipamentos de comunicação usam RTOSs para garantir tempos de resposta rápidos e confiáveis.
3. **Comunicação e Redes**: Em sistemas de comunicação e redes, os RTOSs ajudam a garantir a entrega oportuna de pacotes de dados e a sincronização de processos de rede.

FreeRTOS:

O FreeRTOS é um dos RTOSs de código aberto mais populares e amplamente utilizados. Ele oferece um kernel RTOS pequeno e altamente portável, que pode ser executado em uma variedade de microcontroladores e microprocessadores. Algumas características específicas do FreeRTOS incluem:

- **Gratuito e de código aberto**: O FreeRTOS é distribuído sob a licença MIT, o que significa que é livre para uso comercial e não requer taxas de licenciamento.

- **Portabilidade**: Pode ser portado para uma ampla gama de arquiteturas de microcontroladores e microprocessadores.
- **Baixo consumo de recursos**: O kernel do FreeRTOS é projetado para ser compacto e eficiente em termos de uso de memória e CPU.
- **Suporte para preempção**: O FreeRTOS suporta preempção, permitindo que tarefas de maior prioridade interrompam tarefas de menor prioridade quando necessário.

Funções Importantes de um RTOS:

- Tasks (Tarefas): As tarefas são unidades básicas de execução em um RTOS. Cada tarefa representa uma sequência de instruções que é executada independentemente das outras tarefas.
- Semaforos: Os semáforos são usados para sincronização e exclusão mútua entre tarefas. Eles garantem que recursos compartilhados sejam acessados de forma segura e evitam condições de corrida.
- Mutexes (Mutex - Mutual Exclusion): Semelhante aos semáforos, os mutexes são usados para garantir a exclusão mútua, permitindo que apenas uma tarefa por vez acesse um recurso compartilhado.
- Filas: As filas são usadas para comunicação entre tarefas. Elas permitem que as tarefas enviem e recebam mensagens de forma segura e assíncrona.

Essas funções são fundamentais para o desenvolvimento de sistemas em tempo real e ajudam a garantir a confiabilidade, determinismo e segurança das aplicações embarcadas e sistemas críticos.

Introdução – Trabalhando com tasks

Em FreeRTOS, uma *task* é uma unidade básica de execução que representa uma sequência de instruções que podem ser executadas de forma independente das outras *tasks* em um sistema. Cada *task* em FreeRTOS possui sua própria pilha e contexto de registro, permitindo que ela execute seu código de maneira isolada e concorrente com outras tarefas no sistema.

Para criar e executar uma *task* em FreeRTOS, é necessário definir uma função que contenha o código a ser executado pela *task* e chamar a função `xTaskCreate()` para criar a *task*. Aqui está um exemplo simples de como criar uma *task* em FreeRTOS:

```
// Protótipo da função que será executada pela task
void vTaskFunction(void *pvParameters);

void setup() {
    // Inicialização do FreeRTOS
    // Criando a task com xTaskCreate
    xTaskCreate(vTaskFunction,   // Função que será executada
                "Task",          // Nome da task (opcional)
                1000,            // Memória reservada
                NULL,            // Parâmetros da task
                1,               // Prioridade da task
                NULL);           // Task Handle

    // É possível criar quantas tasks forem necessárias

    // Inicialização do sistema FreeRTOS
    vTaskStartScheduler();
}

void loop() {
    // O loop não é utilizado quando se utiliza FreeRTOS
}

// Função que será executada pela task
void vTaskFunction(void *pvParameters) {
    while(1) {
        // Código da task
    }
}
```

Neste exemplo, uma *task* é criada usando a função `xTaskCreate()`. A função `vTaskFunction()` é a função que será executada pela *task*. Dentro desta função, você pode implementar o código que deseja que a *task* execute.

É importante notar que, uma vez que o scheduler do FreeRTOS é iniciado com `vTaskStartScheduler()`, as tarefas criadas começarão a serem executadas conforme suas prioridades e o algoritmo de escalonamento do sistema. Cada *task* em FreeRTOS é executada até sua conclusão ou até que seja bloqueada por algum evento, como uma operação de E/S ou um temporizador expirado.

Gerenciamento de memória

Se uma task ficar sem memória durante o runtime, todo seu programa falhará e a esp32 fará um reboot. Uma aviso de erro será enviado pela serial se isso ocorrer.

A função `UBaseType_t uxTaskGetStackHighWaterMark(TaskHandle_t xTask);` retorna a quantidade de bytes disponível em uma task. Utilize ela como forma de debug durante a fase de projeto.

Queues

As filas são a principal forma de comunicação entre tarefas. Eles podem ser usadas para enviar mensagens entre tarefas e entre interrupções e tarefas. Na maioria dos casos, elas são usados como buffers FIFO (First In First Out) seguros para threads, com novos dados sendo enviados para o final da fila, embora os dados também possam ser enviados para a frente.

A forma de criar uma fila é a seguinte:

```
xQueueHandle xQueue;

int main(void){
      xQueue = xQueueCreate(5, sizeof(long));

      if(xQueue != NULL)
}
```

Funções para gravar dados na fila:

```
portBASE_TYPE xQueueSendToFront(   xQueueHandle xQueue,
                                   const void* pvItemToQueue,
                                   portTickType xTicksToWait
                                );
portBASE_TYPE xQueueSendToBack(    xQueueHandle xQueue,
                                   const void* pvItemToQueue,
                                   portTickType xTicksToWait
                                );
```

xTicksToWait define o tempo que a task pode esperar para que seja liberado um espaço na fila. O tempo definido por xTicksToWait é dado em milissegundos.

```
long lValueToSend = 10;
portBASE_TYPE xStatus;  // pdTRUE ou pdFALSE
xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0);
```

Exemplo de leitura:

```
xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
if( xStatus == pdPASS) {
      vPrintStringAndNumber( “Received = ”, lReceivedValue );
} else {
      // Pode acontecer da lista estar vazia na leitura
      vPrintString(“Could not receive from the queue.\n” );
}
```

Exemplo: retorna a quantidade de elementos presentes na fila:

```
if( uxQueueMessagesWaiting( xQueue ) != 0 ){
      vPrintString( “Queue should have been empty!\n” );
}
```

Mutex

Em FreeRTOS, os mutexes são usados para controlar o acesso concorrente a recursos compartilhados em um sistema embarcado multitarefa. Um mutex, ou "mutual exclusion semaphore", é basicamente um semáforo binário usado para sincronização entre tarefas. Ele permite que apenas uma tarefa por vez acesse o recurso protegido pelo mutex.

Aqui está uma visão geral básica de como os mutexes são usados em FreeRTOS:

1. **Criação de Mutex:**

Para criar um mutex em FreeRTOS, você normalmente usaria a função `xSemaphoreCreateMutex()` ou `xSemaphoreCreateMutexStatic()`. A primeira aloca dinamicamente a memória para o mutex, enquanto a segunda permite que você forneça uma área de armazenamento estática para o mutex.

2. **Obtenção de Mutex (Lock):**

Uma tarefa obtém o acesso ao recurso protegido pelo mutex chamando a função `xSemaphoreTake()` ou `xSemaphoreTakeRecursive()`. Se o mutex estiver disponível, a função retorna imediatamente, e a tarefa pode prosseguir. Se o mutex estiver atualmente sendo usado por outra tarefa, a tarefa solicitante será colocada em um estado de espera até que o mutex esteja disponível.

3. **Liberação de Mutex (Unlock):**

Quando uma tarefa termina de usar o recurso protegido pelo mutex, ela libera o mutex chamando `xSemaphoreGive()` ou `xSemaphoreGiveRecursive()`. Isso permite que outras tarefas, que estavam esperando pelo mutex, o obtenham e acessem o recurso.

É importante notar que o uso correto dos mutexes é fundamental para evitar condições de corrida e garantir a consistência dos dados compartilhados em sistemas multitarefa. Além disso, o tempo durante o qual um mutex é mantido deve ser mantido o mais curto possível para minimizar o bloqueio de outras tarefas.

Um exemplo simples de uso de mutex em FreeRTOS pode ser algo assim:

```
// Criação do Mutex
SemaphoreHandle_t xMutex = xSemaphoreCreateMutex();

void Task(void *pvParameters) {
    // Tentativa de obter o Mutex
    if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
        // Seção crítica, acesso ao recurso protegido pelo mutex

        // Liberação do Mutex quando a seção crítica for concluída
        xSemaphoreGive(xMutex);
    }
    vTaskDelete(NULL);
}
```

Neste exemplo, a seção crítica é protegida pelo mutex. A tarefa tenta obter o mutex usando `xSemaphoreTake()`. Se o mutex estiver disponível, a tarefa entra na seção crítica, realiza seu trabalho e, em seguida, libera o mutex usando `xSemaphoreGive()`. Se o mutex não estiver disponível, a tarefa ficará bloqueada até que o mutex seja liberado por outra tarefa.