Apuntes - 3° Bimestre

Sistemas embebidos

TIC - 2021

Apuntes - 3° Bimestre

Temario

- Maquinas de estados
- Diagramas de estados
- Timers
- Interrupciones

Material complementario

Recursos principales

Arquitectura de Software Embebido - Universidad de Michigan

Timers STM32 - Documentación

Interrupciones Externas STM32 - Documentación

Otros recursos

Comparación de Arquitecturas de Software Embebido

Timer Arduino Uno - Datasheet

Introducción

En los sistemas embebidos los componentes están pensados específicamente para la tarea que se llevará a cabo. Esto lleva a que los recursos sean limitados y la optimización sea clave en el funcionamiento de los mismos.

Por eso se crearon diferentes formas de administración y manejo de recursos para este tipo de sistemas. Uno de las principales formas son las máquinas de estados.

Máquinas de estados

Las máquinas de estados o state machines, son un modelo de comportamiento de un sistema que colabora con la simplificación y comprensión del mismo.

Las máquinas de estados nos permiten:

- Estandarizar la forma de aproximación y resolución de sistemas.
- Optimizar el manejo de recursos ante la falta de un sistema operativo.
- Evitar los funcionamientos bloqueantes, utilización los recursos solo en los momentos necesarios.
- Compartimentar problemas complejos en partes más simples.

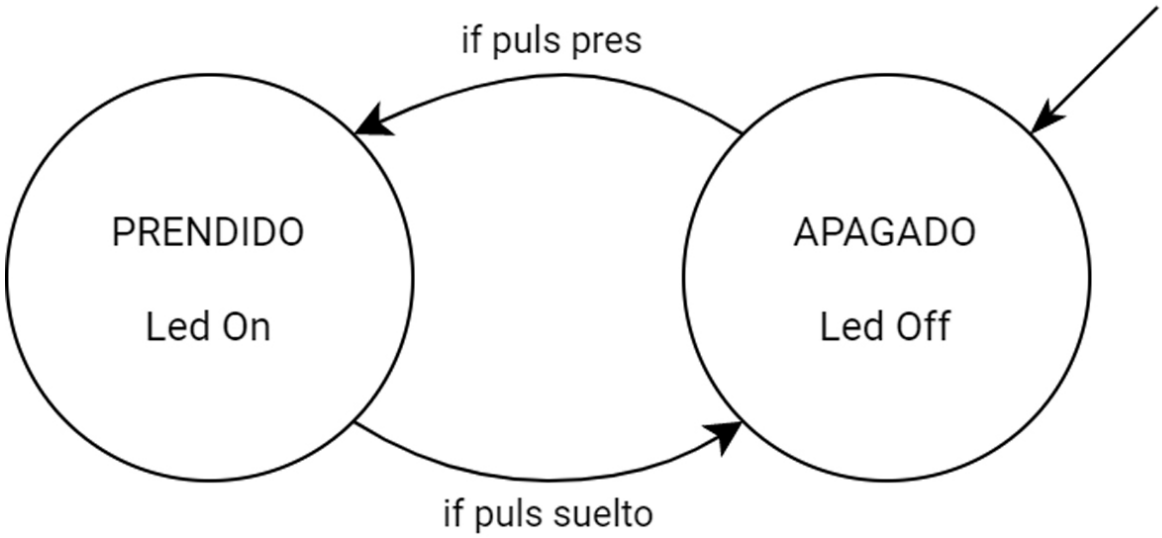
Toda MdE (máquina de estado) está definida por: una cantidad finita de estados, un estado inicial y los desencadenantes de las transiciones.

Diagramas de estados

Los diagramas de estado se utilizan para representar las MdE de una forma visual. Se utiliza un circulo para cada estado, flechas para las transiciones y las condiciones de transición sobre la flecha.

Ejemplo: Utilizar un pulsador para encender y apagar un led.

Se plantea el siguiente diagrama de estado:



Programación de máquinas de estado

Una vez conseguido el diagrama que corresponda a la consigna, se deberá programar el sistema embebido. Para eso utilizaremos defines para los estados y la estructura switch para comparar los diferentes defines con la variable de estado.

Siguiendo el ejemplo anterior, la programación sería la siguiente.

```
//DEFINES DE ESTADOS
// Los valores de estos estados pueden ser aleatorios mientras que sean
diferentes entre sí
#define PRENDIDO 0
#define APAGADO  1

//DEFINES DE PINES
#define LED     12
#define PULS    5

//VARIABLE DE ESTADO
int estado = APAGADO;

//VARIABLES

//INICIALIZACIÓN DE PINES
void setup(){
    pinMode(LED, OUTPUT);
    pinMode(PULS, INPUT_PULLUP);
}

void loop(){
    switch(estado){
        case APAGADO:
            //TAREAS DE ESTADO
            digitalWrite(LED, LOW);
            //DESENCADENADOR-TRIGGER
            if (digitalRead(PULS) == 0)
                estado = PRENDIDO;
            break;
        case PRENDIDO:
            //TAREAS DE ESTADO
            digitalWrite(LED, HIGH);
            //DESENCADENADOR-TRIGGER
            if (digitalRead(PULS) == 1)
                estado = APAGADO;
            break;
    }
}
```

Acá podrán encontrar una plantilla de código Arduino para máquinas de estados.

Arquitecturas de Software Embebido

> ==Software architecture== [...] is defined as the “fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.”

La arquitectura de software embebido se basa en la forma de organización del software para el sistema deseado.

Existen varias arquitecturas diferentes, las principales son las siguientes:

- Round robin
- Round robin with interrupts
- Function queue scheduling
- Real time operating systems (RTOS)

Para poder definir que tipo de arquitectura utilizar se deberán tomar en cuenta varias variables, entre ellas la velocidad de respuesta y la prioridad de las tareas.

Velocidad de respuesta

> Usually, the critical aspect of an embedded control system is its ==speed of response== which is a function of (among other things) the processor speed and the number and complexity of the tasks to be accomplished, as well as the software architecture.
>
> Clearly, embedded systems with not much to do, and plenty of time in which to do it, can employ a simple software organization (a vending machine, for example, or the power seat in your car). Systems that must respond rapidly to many different events with hard real-time deadlines generally require a more complex software architecture (the avionics systems in an aircraft, engine and transmission control, traction control and antilock brakes in your car).

Todo sistema embebido tendrá una velocidad de respuesta, en la que se basará la decisión de que arquitectura implementar.

Un sistema con poco procesamiento y mucho tiempo para hacerlo, podrá utilizar una organización simple. En cambio un sistema que deba responder a muchos eventos en tiempo real, requerirá de una arquitectura más compleja.

Prioridades de tareas

> Most often, the various tasks managed by an embedded system ==have different *priorities*==: Some things have to be done immediately (fire the spark plug precisely 20◦ before the piston reaches top-dead-center in the cylinder), while other tasks may have less severe time constraints (read and store the ambient temperature for use in a calculation to be done later).

Generalmente las tareas a realizar tendrán diferentes prioridades que también deberán ser consideradas al momento de implementar la arquitectura del software.

Arquitectura Round Robin

> The simplest possible software architecture is called “round robin.” ==Round robin== architecture has no interrupts; the software organization ==consists of one main loop wherein the processor simply polls each attached device in turn==, and provides service if any is required. After all devices have been serviced, start over from the top.

> One can think of many examples where round robin ==is a perfectly capable architecture==: A vending machine, ATM, or household appliance such as a microwave oven (check for a button push, decrement timer, update display and start over). Basically, anything ==where the processor has plenty of time to get around the loop, and the user won't notice the delay== (usually micro-seconds) ==between a request for service and the processor response== (the time between pushing a button on your microwave and the update of the display, for example).
>
> [...] there are several obvious disadvantages. ==If a device has to be serviced in less time than it takes the processor to get around the loop, then it won't work.== In fact, the worst case response time for round robin is the sum of the execution times for all of the task code.
>
> Some additional performance can be coaxed from the round robin architecture, however. ==If one or more tasks have more stringent deadlines than the others== (they have higher priority), ==they may simply be checked more often==

La arquitectura Round Robin es la más simple. Consiste de un loop principal donde las tareas se van consultando y ejecutando secuencialmente.

Se puede observar que esta arquitectura puede ser utilizada para complementar una máquina de estados.

Por otro lado, se pueden observar muchas desventajas. Si un dispositivo o componente debe ser atendido antes del tiempo de respuesta del loop, este tipo de arquitectura deja de servir.

En caso de que las tareas tengan prioridades, se pueden chequear más o menos veces que las otras. Esta es una solución simple pero no es la mejor.

Arquitectura Round Robin with interrupts

> Round robin is simple, but that's pretty much its only advantage. One step up on the performance scale is round robin with interrupts. Here, ==urgent tasks get handled in an interrupt service routine==, possibly with a flag set for follow-up processing in the main loop. ==If nothing urgent happens== (emergency stop button pushed, or intruder detected), ==then the processor continues to operate round robin, managing more mundane tasks in order around the loop.==
>
> [...] if the interrupted low priority function is in the middle of a calculation using data that are supplied or modified by the high priority interrupting function, care must be taken that on the return from interrupt the low priority function data are still valid (by disabling interrupts around critical code sections, for example).

La arquitectura Round Robin with interrupts (con interrupciones) es una mejora de la Round Robin, que utiliza las interrupciones para dar prioridad a las tareas de alta prioridad. Mientras que no haya ninguna interrupción, el ciclo se comporta simplemente como un Round Robin.

Al incluir interrupciones es importante tomar en consideración que partes del código funcionan si son interrumpidas y actuar acorde a ello. Por ejemplo, deshabilitar las interrupciones al rededor de la recepción de valores de un sensor, ya que al volver al interrumpirlo podría devolver valores basura.

Arquitectura Function Queue Scheduling

> Function queue scheduling ==provides a method of assigning priorities to interrupts==. In this architecture, interrupt service routines accomplish urgent processing from interrupting devices, but then put a pointer to 5 a handler function on a queue for follow-up processing. The main loop simply checks the function queue, and if it's not empty, calls the first function on the queue. Priorities are assigned by the order of the function in the queue – there's no reason that functions have to be placed in the queue in the order in which the interrupt occurred.
>
> In the Function Queue Scheduling architecture, interrupt routines add function pointers to a queue of function pointers. The main program calls the function pointers one at a time based on their priority in the queue.

Este tipo de arquitectura proporciona una forma de asignar prioridades a las interrupciones. Las interrupciones se utilizarán solamente para agregar a la cola de acciones la próxima actividad a hacer, colocándola en la posición correspondiente a su prioridad.

Arquitectura Real-time Operating System (RTOS)

> A real-time operating system is complicated, potentially expensive, and takes up precious memory in our almost always cost and memory constrained embedded system. Why use one? There are two main reasons: ==flexibility and response time==.
>
> The RTOS schedules when each task is to run based on its priority. ==The scheduling of tasks by the RTOS is referred to as multi-tasking.== In a preemptive multi-tasking system, the RTOS can suspend a low priority task at any time to execute a higher priority one, consequently, the worst case response time for a high priority task is almost zero (in a non-preemptive multi-tasking system, the low priority task finishes executing before the high priority task starts). In the simplest RTOS, a task can be in one of three states:
>
> ==The part of the RTOS called a scheduler keeps track of the state of each task, and decides which one should be running.== The scheduler is a simple-minded device: It simply looks at all the tasks in the ready state and chooses the one with the highest priority.

En este tipo de arquitectura se cuenta con un sistema operativo que decidirá que y cuando ejecutar las tareas mientras que el desarrollador solamente asigna prioridades a las mimas.

Al contar con más de un núcleo se genera la posibilidad de ejecutar varias tareas en paralelo, a lo que llama multi-tasking. Esto aumenta la flexibilidad y el tiempo de respuesta del sistema.

Todo RTOS tiene un planificador o scheduler que se encarga de mantener registro de los estados de cada tarea y decidir cuando ejecutarlas.

Cuando se cuenta con un solo núcleo no existe la posibilidad de hacer multitasking por lo que el RTOS pasa a ser simplemente un scheduler de tareas, muy similar al Function Queue Scheduler. Pero con la simpleza de tercerizarlo mediante una librería. Un RTOS muy utilizado para este tipo de controladores que se utilizan en sistemas embebidos es el FreeRTOS.

Timers y counters

Un **timer** es un tipo de reloj que se utiliza para medir intervalos de tiempo.

Un **contador** es un dispositivo que se encarga de almacenar la cantidad de veces que sucedió un evento o proceso, utilizando de referencia una señal de tiempo.

En general los microcontroladores suelen contar por lo menos con un timer/counter de propósito general que puede ser configurado y utilizado acorde a la tarea necesaria.

El manejo de los timers internamente suele ser bastante complejo, por lo que existen varias librerías para simplificarlo. Una de las librerías más comunes en Arduino es la librería Timer One, que permite el manejo de el Timer 1 del Arduino Uno. Para el STM, las librerías ya vienen incluidas y la documentación es la siguiente. Se puede observar que contamos con 4 timers disponibles para configurar y utilizar.

Cómo funcionan

Un timer se utilizará para medir tiempos mediante interrupciones. Es decir que se va a estar ejecutando el loop, cada cierto tiempo se interrumpirá para ejecutar la acción del timer y luego volver al loop.

Una de las ventajas principales es que esto permite que el código no sea bloqueante.

Parámetros

Los timers tienen diferentes parámetros que pueden ser configurados según la necesidad del sistema.

- **Período.** Tiempo entre interrupciones.
- **Función de interrupción.** Función que se ejecutará luego de cada interrupción.
- **Modo.** En el STM los timers tienen dos modos posibles: comparadores y PWM.

Programar Timers

Configuración

> In order to use timer interrupts, we recommend the following sequence:
>
> - Pause the timer.
> - Configure the prescaler and overflow.
> - Pick a timer channel to handle the interrupt and set the channel's *mode* to `TIMER_OUTPUT_COMPARE`.
> - Set the channel compare value appropriately (this controls what counter value, from 0 to overflow - 1). If you just want to make the interrupt fire once every time the timer overflows, and you don't care what the timer count is, the channel compare value can just be 1.
> - Attach an interrupt handler to the channel.
> - Refresh the timer.
> - Resume the timer.

La secuencia para **configurar timers** es la siguiente:

1. **Pausar** todos los timers a utilizar
2. Setear los **periodos** de interrupción
3. Setear los **modos** de interrupción
4. Setear el **momento de interrupción**, en 0 (al iniciar el contador) o en -1 (al llegar al máximo)
5. Asociar las **funciones** a interrumpir
6. **Refrescar** las interrupciones
7. **Resumir** las interrupciones

Para el STM quedaría algo como lo siguiente:

```
void setup() {
  //SETUP TIMER1
  Timer1.pause();
  Timer1.setPeriod(500000); // in microseconds. Ej: 500000 = 500 milisec
  Timer1.setMode(TIMER_CH1,TIMER_OUTPUT_COMPARE);
  Timer1.setCompare(TIMER_CH1, 1);  // Interrupts on overflow
  Timer1.attachInterrupt(TIMER_CH1, interruptTimer);
  Timer1.refresh();
  Timer1.resume();
}

void loop(){
    //TAREAS DEL LOOP
}

void interruptTimer(){
    //TAREAS A INTERRUMPIR
}
```

Utilización

Una vez configurado el timer se pueden utilizar las siguientes funciones para darles diferentes funcionamientos:

- La función `refresh` hace que nuestro timer vuelva a 0.
- La función `pause` hace que el timer pause la cuenta.
- La función `resume` hace que el timer vuelva a contar.

Ejemplo

Suponiendo que nuestro sistema es un blink de un led cada 500 milisegundos, podemos proponer la siguiente máquina de estados:

Que luego se programará de la siguiente forma:

```
#define APAGADO   0
#define PRENDIDO  1

#define LED 9

int estado = APAGADO;

void setup() {
  pinMode(LED, OUTPUT);

  Timer1.pause();
  Timer1.setPeriod(500000); // in microseconds
  Timer1.setMode(TIMER_CH1, TIMER_OUTPUT_COMPARE);
  Timer1.setCompare(TIMER_CH1, -1);  // Interrupts on overflow
  Timer1.attachInterrupt(TIMER_CH1, interruptTimer);
  Timer1.refresh();
  Timer1.resume();
}

void loop() {
```

```
  switch (estado)
  {
    case PRENDIDO:
      digitalWrite(LED, HIGH);
      break;

    case APAGADO:
      digitalWrite(LED, LOW);
      break;
  }
}

void interruptTimer(void)
{
  if (estado == PRENDIDO)
    estado = APAGADO;
  else
    estado = PRENDIDO;
}
```

Interrupciones externas

Otra forma de interrupción más allá de los timers son las interrupciones externas. Se llama así a las interrupciones generadas por entradas externas al microcontrolador, es decir entradas de señales generadas por pines.

Como funcionan

El concepto se mantiene igual: el programa se está ejecutando dentro del loop y al activar una entrada (por ejemplo un pulsador) se genera una interrupción, para luego realizar la función deseada, y terminar volviendo al loop en la posición donde estaba.

Parámetros

- **Pin.** Número de pin asociado a la entrada.
- **Función de interrupción.** Función que se ejecutará luego de cada interrupción.
- **Modo.** Modo de interrupción:
  - RISING: To trigger an interrupt when the pin transitions LOW to HIGH.
  - FALLING: To trigger an interrupt when the pin transitions HIGH to LOW.
  - CHANGE: To trigger an interrupt when the pin transitions from LOW to HIGH or HIGH to LOW (i.e., when the pin changes).

Programar Interrupciones externas

La programación de interrupciones externas es mucho más simple que la de los timers. Simplemente se debe configurar en el setup y crear una función a ejecutar por la interrupción.

Por ejemplo, si queremos prender y apagar un LED solamente cuando el pulsador terminó de presionarse, se haría de la siguiente manera:

```
#define APAGADO   0
#define PRENDIDO  1

#define LED   PB9
#define PULS  PB12

int estado = APAGADO;

void setup() {
  pinMode(LED, OUTPUT);
  pinMode(PULS, INPUT_PULLUP);

  //CONFIGURACIÓN DE LA INTERRUPCIÓN EXTERNA
  attachInterrupt(PULS, extInterrupt, RISING);
}

void loop() {
  switch (estado)
  {
    case PRENDIDO:
      digitalWrite(LED, HIGH);
      break;
```

```
    case APAGADO:
      digitalWrite(LED, LOW);
      break;
  }
}

//FUNCIÓN DE INTERRUPCIÓN
void extInterrupt(void)
{
  if(estado == PRENDIDO)
      estado = APAGADO;
  else
      estado = PRENDIDO;
}
```

Pinout SHIELD

```
#define LED1  PB4
#define LED2  PB5
#define LED3  PB6
#define LED4  PB7
#define LED5  PB8
#define LED6  PB9

#define POT   PA0

#define LDR   PA1

#define PUL1  PB12
#define PUL2  PB13

#define SW1   PB15
#define SW2   PB14
```

Consideraciones al programar para STM

LED 1 No funciona

Debido a un error interno de una de las librerías de STM, el LED 1 no funciona. Para solucionarlo, después de haberlo definido como una salida deberemos colocar la siguiente línea de programación.

```
afio_cfg_debug_ports(AFIO_DEBUG_SW_ONLY);
```

Quedando algo como lo siguiente:

```
void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(LED4, OUTPUT);
  pinMode(LED5, OUTPUT);
  pinMode(LED6, OUTPUT);
  afio_cfg_debug_ports(AFIO_DEBUG_SW_ONLY);

  //Todos los otros defines
}
```

Timer interrumpe antes de tiempo

Una de las configuraciones del timer permite decidir si el timer interrumpe al iniciar a contar, o al llegar al máximo.

Probablemente si está interrumpiendo antes de tiempo, se solucione colocando el `setCompare` en -1 (al terminar de contar). Esto se hace de la siguiente manera.

```
void setup(){
  //DEFINES DE PINES

  //INICIALIZACIONES DE TIMER
  Timer2.setCompare(TIMER_CH1, -1);
  //RESTO DE INICIALIZACIONES
}
```

Micaela Viegas