



Sistemas Embebidos

Diagrama de Estado – Codificación en C



Laboratorio de
Sistemas Embebidos

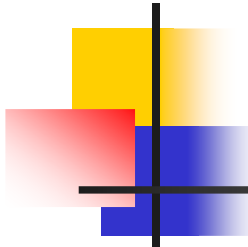
[http://laboratorios.fi.uba.ar/lse/
seminario-embebidos@googlegroups.com](http://laboratorios.fi.uba.ar/lse/seminario-embebidos@googlegroups.com)

66.48 & 66.66 Seminario de Electrónica: Sistemas Embebidos

Curso de Posgrado: Introducción a los Sistemas Embebidos

Ingeniería en Electrónica – FI – UBA

Buenos Aires, 18 de Octubre de 2011



Temario

- Métodos de codificación en C
 - Switch (múltiples if)
 - Punteros a función
 - Tabla de Estado
 - Patrones de diseño de Estado orientados a objetos
 - Otras técnicas que combinan a las anteriores (frameworks)

Manejador de Salidas (EJ-01-a)

- Control de LED con estado **Fijo** manejado por **evento** (dos estados simples)

led



LED

APAGADO

ENCEND.

- Eventos

- eCambio

→ para cambiar de estado

- Acciones

- aApagar
- aEncender

→ para apagar el LED

→ para encender el LED

- Estados

- APAGADO
- ENCENDIDO

→ mientras se desee que esté apagado

→ mientras se desee que esté encendido



Con Switch

- Declarar los **estados, eventos, variables y acciones**
- Implementar la inicialización del diagrama de estado
- Implementar el diagrama de estado comenzando por validar la consistencia de la variable **estado** y luego abrir un **switch**
- La variable de control del **switch** es la variable de **estado** del diagrama de estados
- En el **switch** habrá un **case** para **cada estado**
- Dentro de cada **case** habrá que establecer una o más **proposiciones** para diferenciar las transiciones salientes de ese estado
- Ordenar las proposiciones en función de las prioridades asignadas a cada transición

Con Switch

/* Es necesario definir los estados y constantes */

```
enum EstadosMELed {APAGADO, ENCENDIDO, ESTADOMAX};
```

```
#define ESTADOiNI    APAGADO
```

```
enum EventosMELed {eCambio};
```

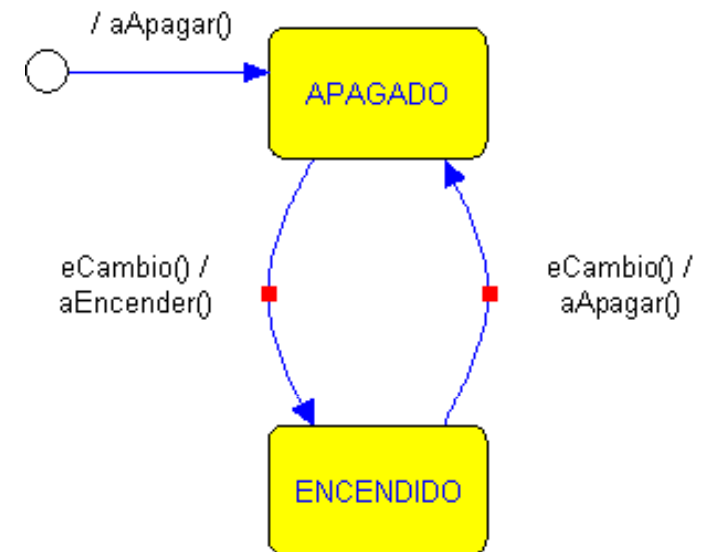
/* Es necesario declarar algunas variables */

```
uint8_t char estado, evento;
```

/* Es necesario declarar e implementar dos funciones */

```
void InicializarMELed (void);
```

```
void MELed (void);
```



Con Switch

/* Se implementa la transición del **Initial** a **APAGADO** */

```
void InicilaizarMELed (void)
```

```
{
```

```
    estado = ESTADOiNI;
```

```
    Apagar ();
```

```
}
```

/* Se implementan el resto de las transiciones salientes de c/estado en función del evento */

```
void MELed (void)
```

```
{
```

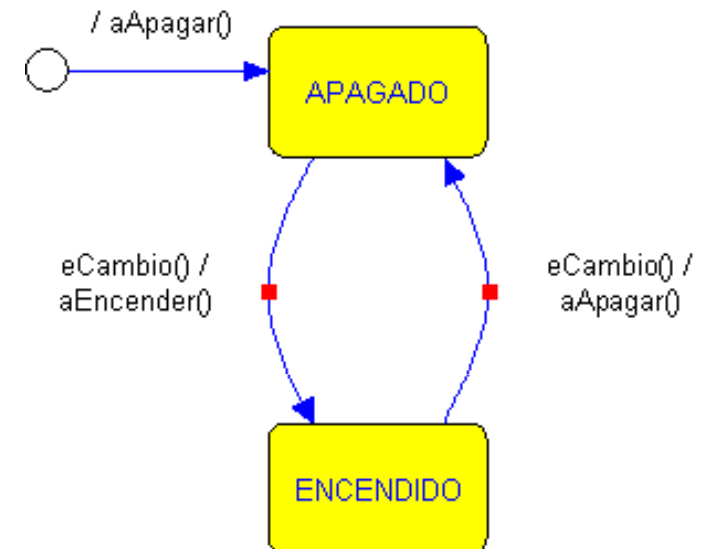
```
    /* Primero valido la variable de estado */
```

```
    if (estado >= ESTADOmAX) {
```

```
        InicializarMELed ();
```

```
        return;
```

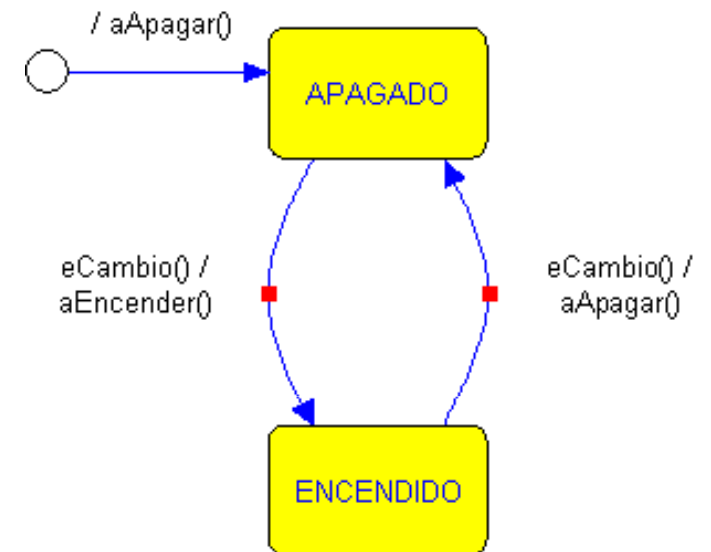
```
    }
```



// Ojo!!!: hacer lo adecuado a la aplicación

Con Switch

```
switch (estado) {  
  case APAGADO:  
    if (evento == eCambio) {  
      estado = ENCENDIDO;  
      Encender ();  
    }  
    break;  
  case ENCENDIDO:  
    if (evento == eCambio) {  
      estado = APAGADO;  
      Apagar ();  
    }  
    break;  
}
```



// Se puede reemplazar el **switch** por múltiples **if**



Enumeración de estados

```
enum EstadosMELed {APAGADO, ENCENDIDO, ESTADOMAX};  
uint8_t estado;
```

VS

```
typedef enum {APAGADO, ENCENDIDO, ESTADOMAX} EstadosMELed;  
EstadosMELed estado;
```

- ¿Cuál es la diferencia?
- ¿Cuál elegiría y por qué?



Valor del Estado Máximo

```
if (estado >= ESTADOmAX) {  
    InicializarMELed ();  
    return;  
} // {APAGADO, ENCENDIDO, ESTADOmAX}
```

VS

```
if (estado > ESTADOmAX) {  
    InicializarMELed ();  
    return;  
} // {APAGADO, ENCENDIDO, ESTADOmAX = ENCENDIDO}
```

- ¿Cuál es la diferencia?
- ¿Cuál elegiría y por qué?



Uso de default del switch

```
if (estado ? ESTADOmAX) {  
    InicializarMELed ();  
    return;  
}
```

VS

```
switch (estado) {  
    .....  
    default:  
        InicializarMELed ();  
    break;    // ¿Con {APAGADO, ENCENDIDO, ESTADOmAX} o con  
              // {APAGADO, ENCENDIDO, ESTADOmAX = ENCENDIDO}?  
}
```

- ¿Cuál es la diferencia?
- ¿Cuál elegiría y porqué?



Observaciones

- Estructuras
 - ¿Sería conveniente definir alguna/s estructura/s?
 - ¿Cuáles definiría y porqué?
- Eventos
 - ¿Sería conveniente definir otro medio para pasar evento a una ME?
 - ¿Cuáles definiría y porqué?
- Exit/Entry/Do
 - ¿La solución puede contener acciones especiales y actividades?
 - ¿Cómo lo haría y porqué?
- Subestados Secuenciales y Concurrentes
 - ¿La solución puede contener Subestado secuenciales y concurrentes?
 - ¿Cómo lo haría y porqué?



Eventos Temporizados

- Un evento temporizado no altera la estructura de la solución vista
- Requiere de una variable del tipo **timerTick** que será administrada por la rutina de atención del Timer de recarga periódica del sistema (que se encargará de decrementarla si es distinta de cero, una vez decrementada detectar si llegó a cero y en tal caso generar el evento de temporizado asociado)
- Mientras que la función de seteo invocada por la máquina de estado se encarga de darle valor inicial

Con punteros a función

/* Es necesario definir los estados, constantes, declarar variables, las función del switch, una función para cada estado y un array de punteros a función */

```
void Apagado (void);
```

```
void Encendido (void);
```

```
void (* const ArrayFuncionesMELed []) (void) = {Apagado , Encendido};
```

```
void MELed (void)
```

```
{
```

```
    if (estado >= ESTADOmAX) {
```

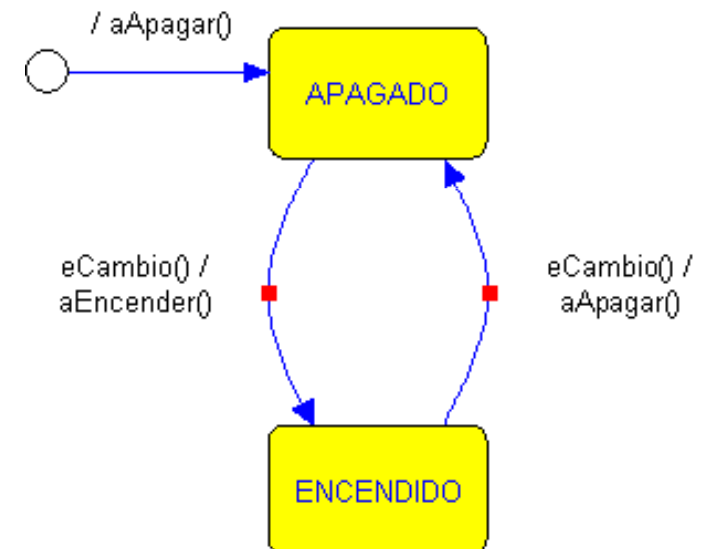
```
        InicializarMELed ();
```

```
        return;
```

```
    }
```

```
    (*ArrayFuncionesMELed [estado]) ();
```

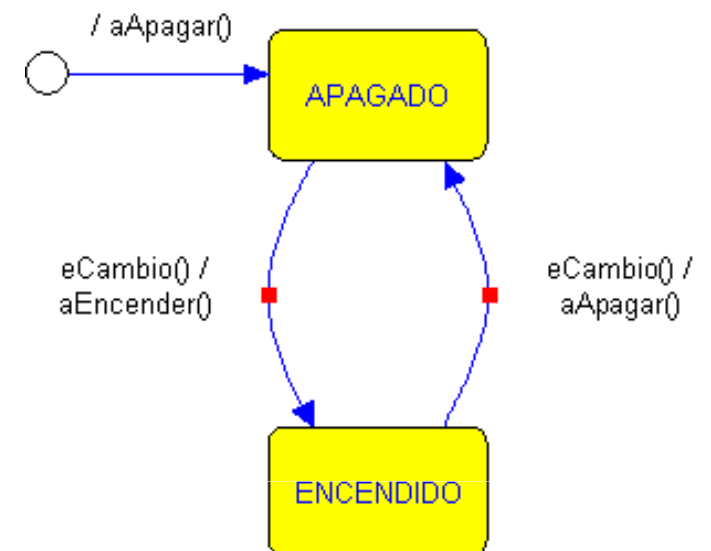
```
}
```



Con punteros a función

```
void Apagado (void)
{
    if (evento == eCambio) {
        estado = ENCENDIDO;
        Encender ();
    }
}
```

```
Void Encendido (void)
{
    if (evento == eCambio) {
        estado = APAGADO;
        Apagar ();
    }
}
```





Switch vs Punteros a Función

- En implementaciones con **switch** los **case** de cada estado se evaluarán secuencialmente, equivale a una cadena **if** consecutivos de resolución "micro & compilador dependiente"
- En implementaciones con **punteros a función** el tiempo de acceso a las funciones es el mismo independientemente del valor de la variable de estado, equivale a un desvío selectivo de la ejecución del programa de resolución "micro & compilador dependiente"
 - Para uniformar aún más este tiempo de acceso podemos reemplazar el **array** por una **matriz** y manejar la 2º dimensión con la variable **evento**
 - Simplificando así las funciones contenidas en la matriz al eliminar de ellas los **if** asociados a la variable **evento** y aumentando la cantidad de funciones al número máximo = filas x columnas de la matriz

Switch & Punteros a Función

- Usualmente cualquiera de estas implementaciones requiere invocar a las dos funciones básicas desde el programa principal (una al inicio y otra en el loop principal)

```
int main (void)
{
    .....
    InicializarMELed ();
    .....
    while (1) {
        .....
        MELed ();
        .....
    }
}
```

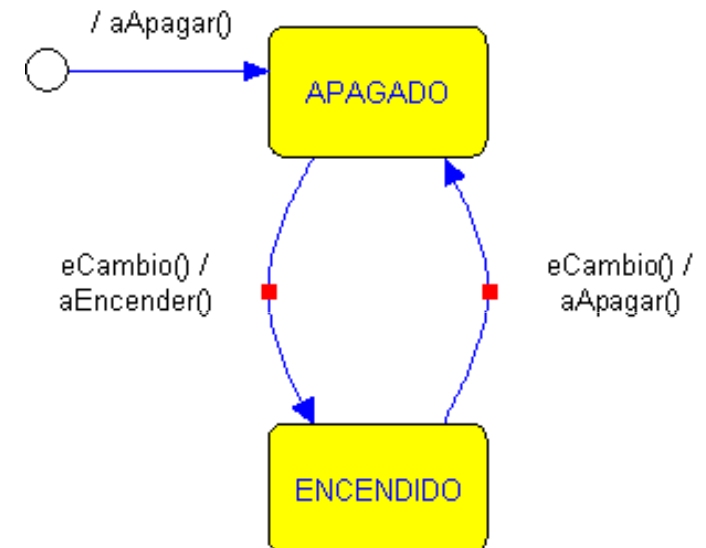




Tabla de Estado

- Una variante de implementación es la de Tabla de Estado, en la que la matriz contiene tanto las funciones como el próximo estado de la máquina de estado

Table 3.1: Two-dimensional state table for the time bomb

		Events →			
		UP	DOWN	ARM	TICK
States →	Setting	<code>setting_UP()</code> , <code>setting</code>	<code>setting_DOWN()</code> , <code>setting</code>	<code>setting_ARM()</code> , <code>timing</code>	<code>empty()</code> , <code>setting</code>
	Timing	<code>timing_UP()</code> , <code>timing</code>	<code>timing_DOWN()</code> , <code>timing</code>	<code>timing_ARM()</code> , <code>setting(*)</code>	<code>timing_TICK()</code> , <code>timing(**)</code>

Notes:

(*) The transition to “setting” is taken only when `(me->code == me->defuse)`.

(**) The self-transition to “timing” is taken only when `(e->fine_time == 0)` and `(me->timeout != 0)`.



Tabla de Estado

Table 3.2: One-dimensional state transition table for the time bomb

Current State	Event (Parameters)	[Guard]	Next State	Actions
setting	UP	[me->timeout < 60]	setting	++me->timeout; BSP_display (me->timeout);
	DOWN	[me->timeout > 1]	setting	--me->timeout; BSP_display (me->timeout);
	ARM		timing	me->code = 0;
	TICK		setting	
timing	UP		timing	me->code <= 1; me->code = 1;
	DOWN		timing	me->code <= 1;
	ARM	[me->code == me->defuse]	setting	
	TICK (fine_time)	[e->fine_time == 0]	choice	--me->timeout; BSP_display (me->timeout);
		[me->timeout == 0]	final	BSP_boom();
		[else]	timing	



Tabla de Estado

- Nótese que en cualesquiera de las implementaciones planteadas hemos tratado de cubrir la totalidad de las transiciones, siendo éstas diversas formas de lograrlo
- Por ello cualquier implementación requiere modificar las funciones básicas de la máquina de estado (**InicializarMELed** y **MELed**)
- Esta modificación se puede hacer mucho más elegante en C recurriendo a estructuras si el controlador para el que estamos desarrollando la aplicación puede manejar direcciones sin problemas ni demoras (generar código compacto y rápido, con mínimo uso de memoria de datos adicional)



Otras implementaciones y TP

- Patrones de diseño de Estado orientados a objetos
- Otras técnicas que combinan a las anteriores (frameworks)
 - "Practical Statecharts in C/C++" by Miro Samek,
<http://www.quantum-leaps.com>
 - "Rhapsody in C++" by ILogix (a code generator solution)
<http://www.ilogix.com/sublevel.aspx?id=53>
 - "State Machine Design in C++" by David Lafreniere,
<http://www.ddj.com/184401236?pgno=1>
 - Boost Software C++ Libraries <http://www.boost.org>
 - Reactive System Framework by Leandro Francucci
<http://sourceforge.net/projects/rkh-reactivesys/>
- En la práctica de modelado se pretende modelar Máquinas de Estado mediante Diagramas de Estado y codificarlas en C para el ambiente de LPCXpresso



Referencias

- Sistemas Embebido 2011 - Diagrama de Estado - J. M. Cruz
- Sistemas Embebido 2011 - Diagrama de Estado - Ejemplos - J. M. Cruz
- Técnicas Digitales II - R4052 - 2011 - Diagrama de Estado - J. M. Cruz
- Practical UML Statecharts in C/C++, 2º Edición - Miro Samek