Implementação de *firmware* como máquina de estados

O código em C disponibilizado em "CPA-Trabalho-state_machine_firmware.rar" simula um firmware construído na forma de máquina de estados finita (MEF) o qual seria utilizado para acionar um LED ligado a uma entrada/saída digital (GPIO - *general purpose input/output*). Esse código deve ser usado como base para a realização da Parte 1 do Trabalho Final da disciplina. A Figura 1 mostra o diagrama do grafo do sistema modelado como um autômato de acordo com a teoria de SEDs vista ao longo do curso.

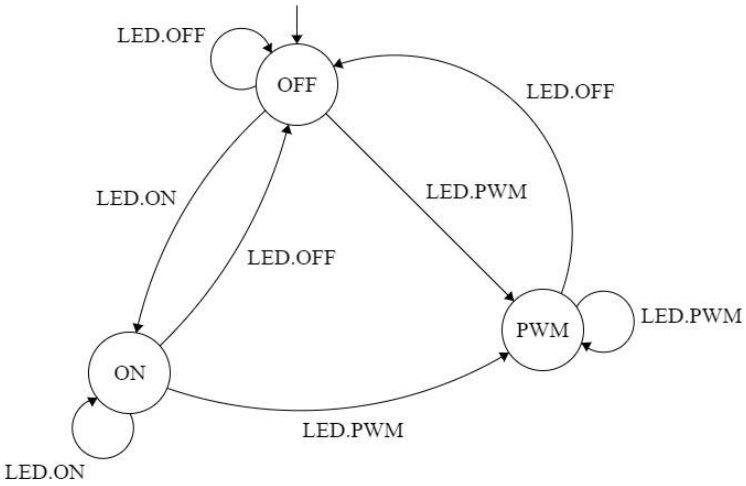

Figura 1

Os eventos descritos são:

- LED.ON: O usuário pressiona um botão para acionar o LED.
- LED.OFF: O usuário pressiona outro botão para acionar o LED.
- LED.PWM: O usuário pressiona outro botão para fazer com o que o LED seja acionado por um sinal PWM. Assume-se que o usuário fornece os parâmetros do PWM (*duty cycle* e frequência) por alguma outra interface não modelada.

Os estados descritos são:

- ON: O LED está ligado.
- OFF: O LED está desligado.
- PWM: O LED pisca de acordo com os parâmetros do PWM.

Deve-se observar que foram adotados dois botões para o acionamento e desligamento do LED, quando um único poderia ter sido usado. Qual seria a diferença? Na implementação do *firmware*, cada estado está associado a uma função do código que é executada a cada ciclo, sempre que um evento é detectado. O período de cada ciclo é muito pequeno, devido à velocidade da CPU. Logo, enquanto o evento "botão pressionado" está ocorrendo, muitos ciclos podem ser executados internamente, o que pode causar instabilidades se um mesmo evento é usado em transições contíguas.

Por exemplo, partindo-se do estado OFF, durante os poucos milissegundos em que o usuário mantém o botão pressionado, o sistema transitaria ininterruptamente de OFF para ON e de ON para OFF, sem que fosse possível determinar qual o estado final após o usuário soltar o botão. Contudo, a ocorrência de instabilidades depende do método de detecção do evento. Se tal método utiliza interrupções externas vinculadas à GPIO do LED, as quais normalmente são disparadas por bordas de subida ou descida aplicadas, não haveria problema em se utilizar o mesmo botão, pois o microcontrolador detectaria, de fato, transições do estado, sendo irrelevante o período durante o qual o botão é mantido pressionado. A instabilidade ocorreria se o método utilizado fosse o *polling*, em que o estado da GPIO seria lido periodicamente. Uma maneira de contornar esse problema seria introduzir um atraso na leitura do estado[1], embora fosse possível ao usuário burlar esse método mantendo o botão pressionado além do tempo de retardo. De outro modo, pode-se simplesmente usar outro botão, como no modelo da Figura 1.

Desta feita, o modelo empregado está sujeito a critérios práticos. Como o objetivo do trabalho é apenas simular uma implementação real, o aluno fica livre para escolher o método que preferir, desde que se atenha às especificações do roteiro. De fato, nas diretivas dadas, de modo geral, é imposta a utilização de duas botoeiras para se ligar e desligar algum elemento do sistema. Quando se trata de CLPs, esse é o método normalmente adotado, o que se justifica pelo fato de tais equipamentos usarem sempre o método de *polling*. Entretanto, ainda com esses dispositivos seria possível contornar tal situação utilizando-se temporizadores ou contadores, embora eles não sejam normalmente utilizados para isso.

Em se tratando do código em C fornecido como exemplo, duas observações devem ser feitas: não há correspondência biunívoca entre os estados do autômato da Figura 1 e os estados da MEF do firmware; e ainda, vários outros métodos de implementação são possíveis e o estudante está livre para experimentar outras possibilidades. A ausência de correspondência biunívoca está associada ao fato de o modelo ocultar demandas relativas ao *hardware* ou outras camadas de código, entre outras complexidades. Logo, para evitar-se a construção de funções extensas, o que prejudicaria a modularidade do código, é preciso quebrar os estados do autômato em 'estados internos' do *firmware*. A Figura 2 ilustra o diagrama da MEF do código fornecido.

Deve-se observar como o estado PWM do autômato foi dividido em dois estados da MEF do firwmare. No primeiro alcançado, o PWM apenas é configurado, no segundo, a comutação da GPIO é realizada periodicamente a cada ciclo de execução da MEF e de acordo com os parâmetros do PWM. É claro que as funções associadas aos dois estados poderiam ser colocadas juntas, uma vez que são simples, e a GPIO poderia ser comutada por uma interrupção disparada por um *timer*. Contudo, o objetivo desse exemplo é ilustrar a diferença entre os estados do autômato e o da MEF do firmware.

Outro ponto importante é que a introdução dos 'estados internos' do firmware exige a definição de eventos internos também. Esses podem ser de dois tipos: bloqueantes e não

[1] Esse atraso seria análogo à histerese propositalmente introduzida em comparadores *Schmitt trigger*. Esse método é comumente empregado em códigos de controle de teclados e botões em geral.

bloqueantes[2]. Os eventos internos bloqueantes não podem ser interrompidos por eventos externos – associados às entradas de sensores, interrupções do usuário, etc. – enquanto os não bloqueantes podem sê-lo. O evento 'int.lock' do autômato da Figura 2 é bloqueante, pois após a configuração do PWM é preciso executá-lo, não admitindo-se intervenções nesse processo. Qualquer evento externo que ocorra em PWM1 é ignorado. Já o evento 'int.unlock' é interno, mas não bloqueante, pois reconhece o evento LED.PWM, que não leva a alteração alguma, e LED.OFF, o que é essencial para que não haja *livelock*. Obviamente que a abordagem adotada poderia ser diferente, como a presença dos laços, por exemplo, que poderiam ser ausentes.

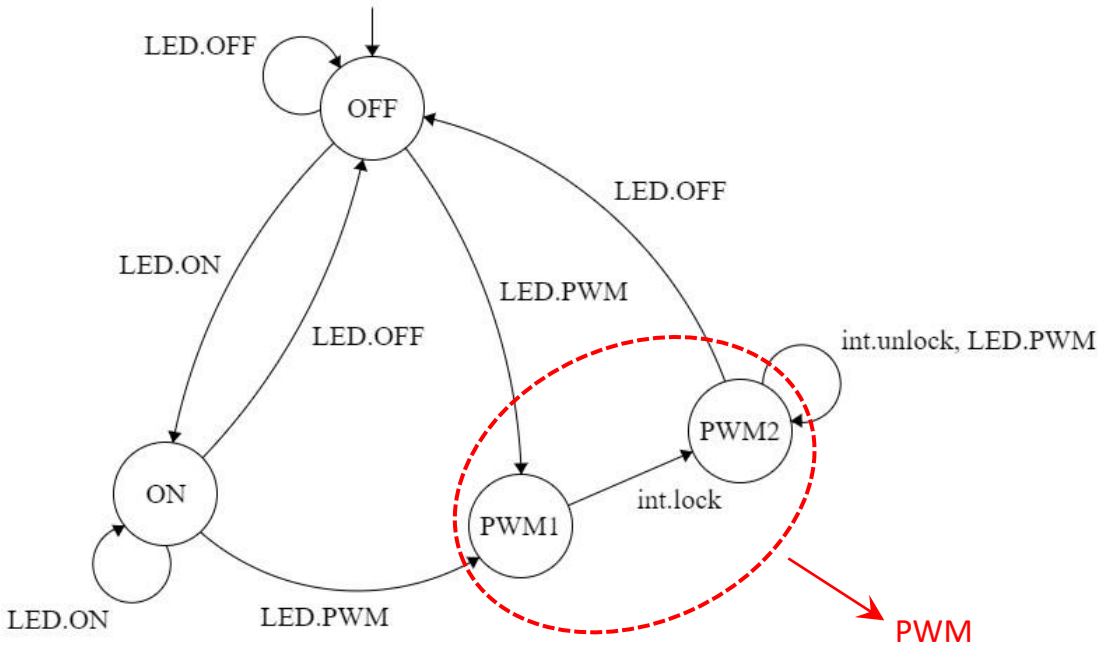

Figura 2

Em relação aos detalhes do código, abaixo são descritos dois tipos criados para representar os estados e eventos. O estudante pode fazer as alterações de acordo com seu modelo. Os valores EV_NONE e ST_NULL, bem como NUMBER_ST e NUMBER_EV, são usados para controle interno. Os estados ST_LED_PWM_SETUP e ST_LED_PWM_RUN correspondem a PWM1 e PW2 na Figura 2.

```
typedef enum
{
   ST_NULL = 0,
   ST_LED_OFF,
   ST_LED_ON,
   ST_LED_PWM_SETUP,
   ST_LED_PWM_RUN,
   NUMBER_ST
} state_id;
```

```
typedef enum
{
   EV_NONE = 0,
   EV_LED_OFF,
   EV_LED_ON,
   EV_LED_PWM,
   EV_INTERNAL,
   EV_INTERNAL_LOCK,
   NUMBER_EV
} event_id;
```

[2] Não se deve confundir esse conceito com o de bloqueio na teoria vista de SEDs. Esse conceito está mais relacionado aos processos não preemptíveis em sistemas operacionais.

Abaixo é exibido o *array* correspondente à função de transição da MEF, onde cada linha é associada a um estado (que seria o atual) e cada coluna a um evento. O elemento presente na posição [estado][evento] é o estado para o qual a MEF deve transitar naquele ciclo. Esse *array* deve ser editado pelo aluno de acordo com seu modelo. Ao fazê-lo, deve-se obedecer à sequência numérica definida por cada evento no tipo event_id.

```
const static state_id f[NUMBER_ST][NUMBER_EV] =
{
   [ST_LED_OFF]       = {ST_NULL, ST_LED_OFF, ST_LED_ON, ST_LED_PWM_SETUP, ST_NULL, ST_NULL},
   [ST_LED_ON]        = {ST_NULL, ST_LED_OFF, ST_LED_ON, ST_LED_PWM_SETUP, ST_NULL, ST_NULL},
   [ST_LED_PWM_SETUP] = {ST_NULL, ST_NULL, ST_NULL, ST_NULL, ST_NULL, ST_LED_PWM_RUN},
   [ST_LED_PWM_RUN]   = {ST_NULL, ST_LED_OFF, ST_NULL, ST_LED_PWM_RUN, ST_LED_PWM_RUN, ST_NULL}
};
```

O código abaixo mostra a definição do *array* de ponteiros das funções responsáveis por executar cada estado da MEF. Esse *array* também deve ser editado por cada aluno, bem como cada função associada. Ele contém elementos da estrutura state_handler, cuja definição é exibida logo em seguida.

```
const static state_handler state_handler_container[NUMBER_ST] =
{
   [ST_LED_OFF] = {ST_LED_OFF, st_led_off},
   [ST_LED_ON]  = {ST_LED_ON, st_led_on},
   [ST_LED_PWM_SETUP] = {ST_LED_PWM_SETUP, st_led_pwm_setup},
   [ST_LED_PWM_RUN] = {ST_LED_PWM_RUN, st_led_pwm_run}
};

typedef struct
{
   state_id st_id;
   int (*state_exe)(STATE_FUNCTIONS_ARGS);
} state_handler;
```

Por fim, cada função é executada num ciclo pela chamada abaixo, onde _state contém o valor do estado atual.

```
ret = state_handler_container[_state].state_exe(ex_data, in_data);
```

Essa técnica de implementação de *firmware* é bastante vantajosa. Entre outros benefícios, ela torna o código facilmente extensível, bastando para isso modificar a função de transição e incorporar as novas funções. Ela também facilita a execução de MEFs distintas e independentes, a despeito da ausência de sistema operacional. Nesse caso, porém, seria preciso gerenciar o acesso aos recursos compartilhados pelas diferentes MEFs.