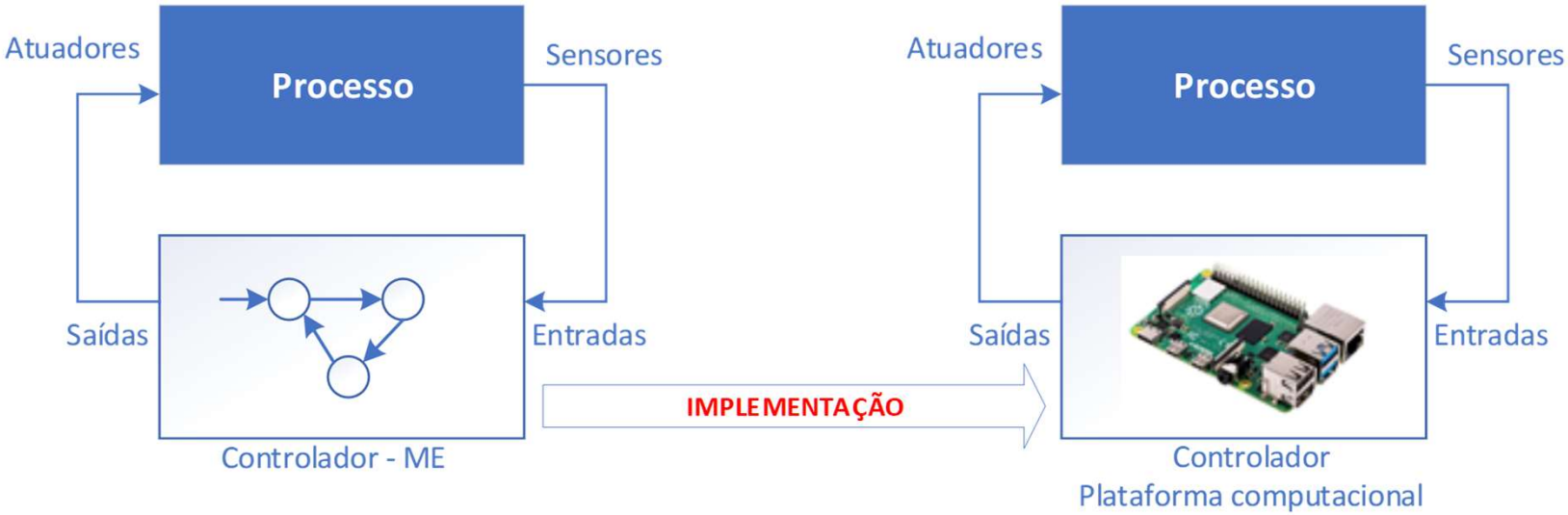
IMPLEMENTAÇÃO DE MÁQUINAS DE ESTADO

Introdução

— Depois de modelar o controlador utilizando máquinas de estado (ME) e assumindo que este foi **validado** (i.e. que cumpre as especificações), é necessário **implementa-lo** numa plataforma computacional.

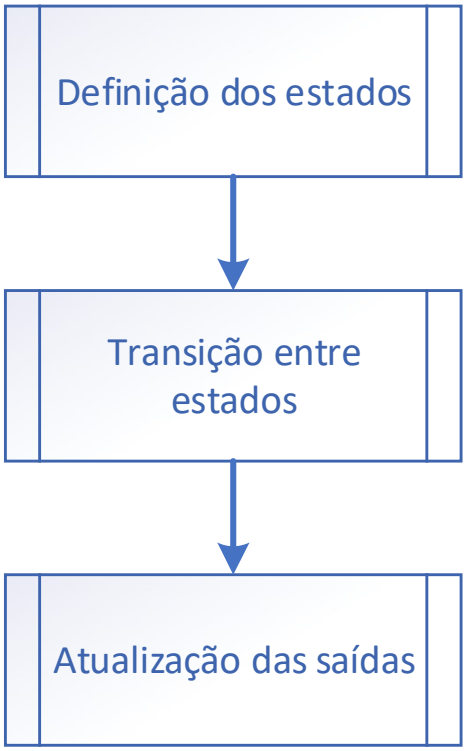


— Vamos apresentar uma **abordagem sistemática** para implementar uma ME numa **plataforma computacional genérica** (ex. microprocessador) utilizando a **linguagem C:**

  — O código apresentado é apenas uma das possibilidades existentes. Existem várias soluções alternativas.

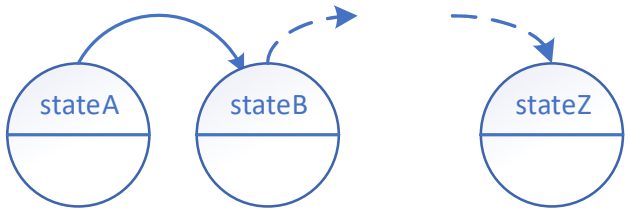
  — O processo seria análogo para qualquer outra linguagem.

Fluxograma de implementação



Definição dos Estados

- Os estados podem ser definidos através uma declaração utilizando um **tipo enumerado:**
  - Cada valor enumerado é interpretado como um inteiro (0, 1,…) em C

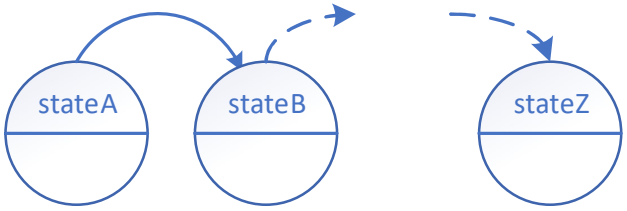


- Deve ser declarada uma **variável** para guardar o estado atual da máquina.
  - A variável deve ser **inicializada** com o **estado inicial**.

```
// Estados da máquina
typedef enum{
     stateA,
     stateB,
     ...
     stateZ
} stateNames;

// Estado atual e inicial da ME
stateNames currentState = stateA;
```

Transições entre estados

— As transições entre estados podem ser implementadas com uma estrutura de decisão **SWITCH….CASE**, utilizando com variável de decisão o estado atual da ME

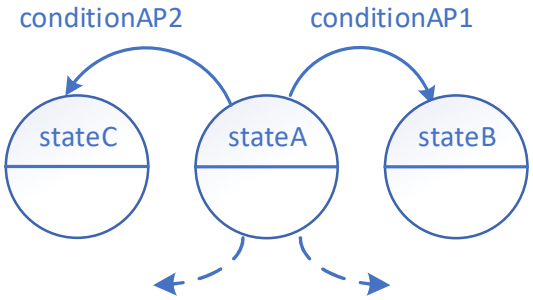


— O **BREAK** permite uma execução mais eficiente, pois só há 1 estado está ativo em cada instante.

```
// Transição entre estados

switch (currentState) {

    case stateA:
    // Transições com origem em stateA
    ...
    break;


    case stateB:
          ...
    // Transições com origem em stateB
    ...
    break;
    ....


    case stateZ:
    // Transições com origem em stateZ
    ...
    break;
}
```

Transições entre estados

— Em cada estado, utilizar uma condição **IF** para testar os **predicados** das respetivas transições e **atualizar** o próximo estado se necessário.



— Ao colocar um **break** em cada transição garante-se que só é processada uma transição (i.e. fica assegurado de imediato a exclusão mutua entre transições)

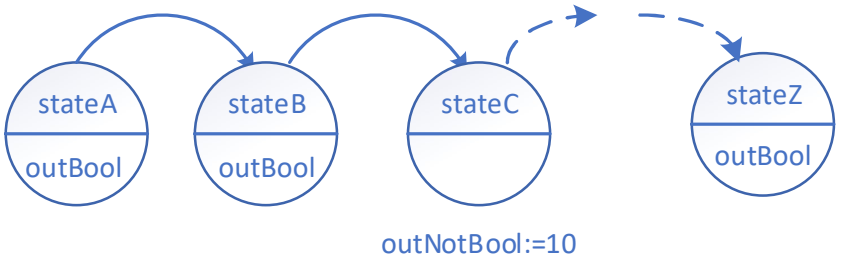
— Com alternativa é possível também utilizar uma estrutura **switch …case**

```
// Transição entre estados
switch (currentState) {

   case stateA:

    // Testa se o predicado associado a uma
    // transição é verdadeiro

    if (conditionAP1) {

          // Próximo estado
          currentState = nextStateB;

          break;
    }
    ...

    if (conditionAP2) {

          // Próximo estado
          currentState = nextStateC;

          break;
    }
    ...


   case stateB
   ...
}
```

Atualização das Saídas

— Saídas **booleanas**: implementadas com uma atribuição a partir da lista de estados em que a saída é colocada a **True**.

  — (**currentState == State_i**) toma o valor **TRUE** quando a ME está no estado **State_i** e **FALSE** caso contrário. O operador **||** (OU) conjuga todos os estados em que a saída deve ser colocada a **TRUE** (nos restantes é **FALSE**)

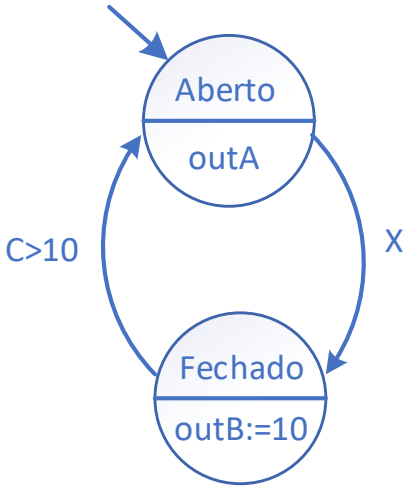


— Saídas **não booleanas**: implementadas através de uma estrutura do tipo **IF**

— As **variáveis internas** da ME podem ser tratadas de forma análoga.

```
// Atualização das saídas



// Saídas booleanas
outBool =  (currentState == stateA) ||
           (currentState == stateB) ||
           (currentState == stateZ);
...









//Saídas não booleanas
if (currentState == stateC) {
           outNotBool = 10;
           ...
}
...
```

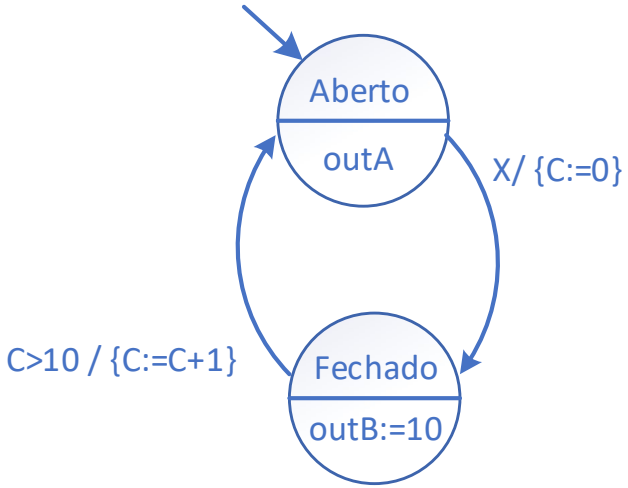
Exemplo




```
// Estados da ME
typedef enum{
     Aberto,
     Fechado
} stateNames;

bool X, outA;
int C, outB;


// Estado atual
stateNames currentState = Aberto;


int main() {

     // Transição entre estados
     switch (currentState) {

          case Aberto :
               if (X)
                    currentState = Fechado;
          break;

          case Fechado :
               if (C > 10)
                    currentState = Aberto;
          break;
     }

     // Saídas booleanas
     outA = (currentState == Aberto);


     //Saídas não booleanas
     if (currentState == Fechado)
          outB = 10;
}
```

Ações associadas a transições

— Na estrutura que implementa a transição entre estados, incluir o código correspondente quando a transição é executada



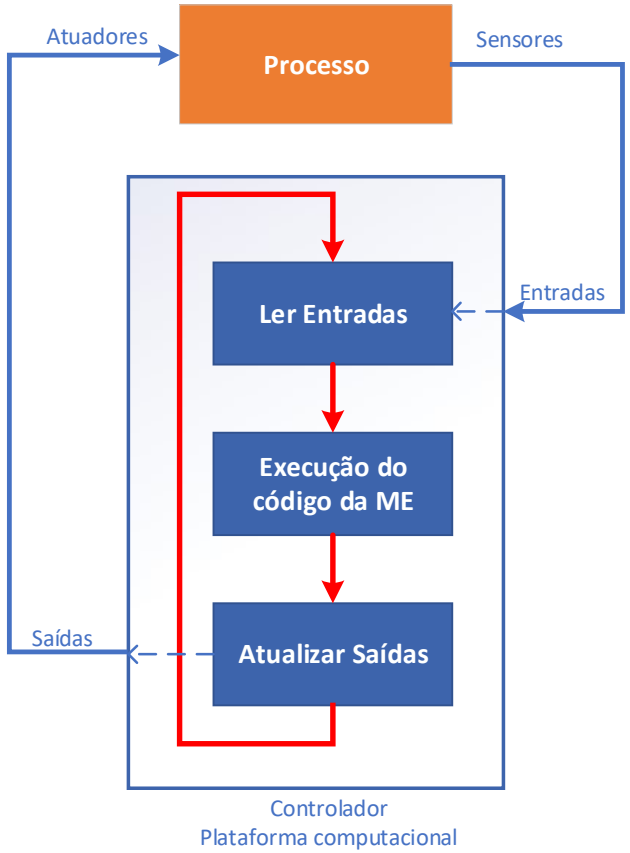

```
// Transição entre estados
switch (currentState) {

    case Aberto :
        if (X){

            // Próximo estado
            currentState = Fechado;

            // Ações associadas a eventos
            C = 0;
        }
    break;

    case Fechado :
        if (C > 10) {

            // Próximo estado
            currentState = Aberto;

            // Ações associadas a eventos
            C = C+1;
        }
    break;
}
```

Discussão

- A implementação do código anterior resultaria **numa única execução:** i.e. o programa executa apenas 1 vez e termina.
- É assim necessário propor uma estrutura que permita uma **execução contínua** do código da máquina de estados.
- As soluções mais comuns para resolver este problema (e que se encontram na maioria dos controladores industriais, tais como autómatos programáveis) são:
  - **Execução Cíclica**
  - **Execução Periódica**

Execução Cíclica & Periódica

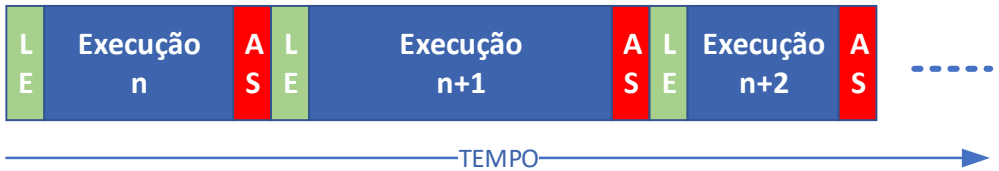
- Características:
  - A execução do código (da ME) é realizada ciclicamente
  - Todas as **entradas são adquiridas no início de cada ciclo.**
    - **Imagem do processo**
  - Todas as **saídas são atualizadas no final de cada ciclo.**
    - **Imagem das saídas.**
- Este tipo de implementação permite:
  - A **sincronização** da leitura de todas as entradas no início de cada ciclo
  - A atualização **simultânea** de todas as saídas no final de cada ciclo

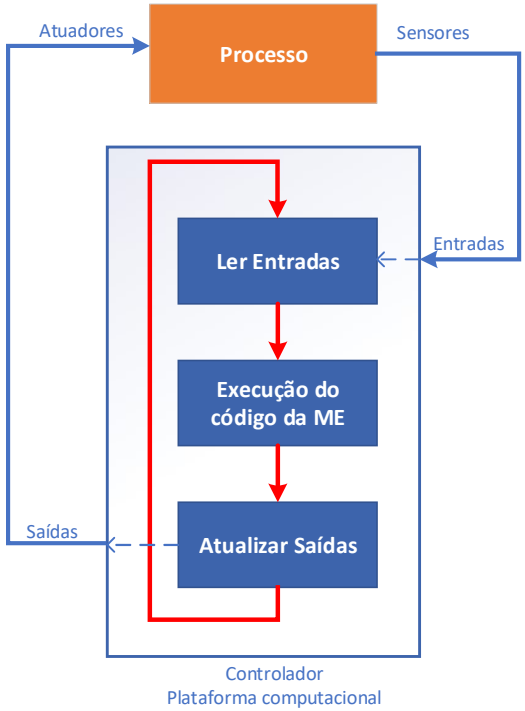


Execução Cíclica

— **Características**

— Quando termina a uma execução, a próxima execução inicia-se de imediato.





— **Vantagens**

— Simples de implementar.

— **Desvantagens**

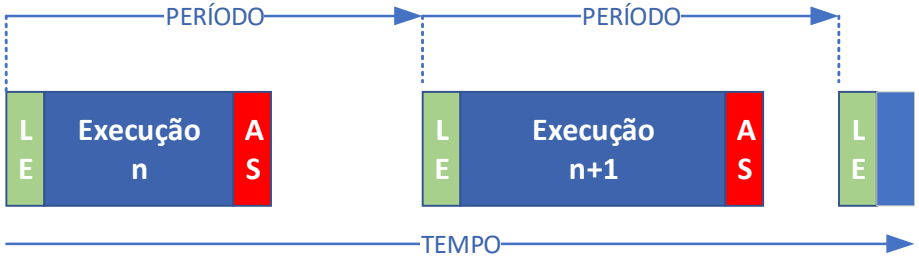
— Não é garantido que a leitura das entradas possa ser realizada com um intervalo constante.

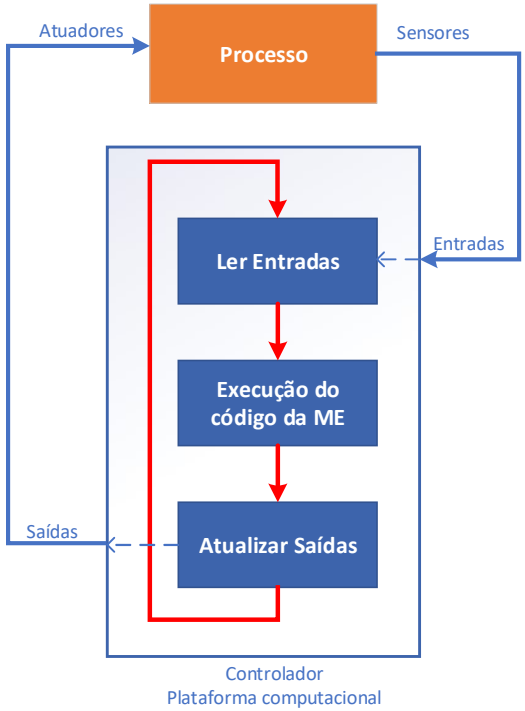
—Os tempos de execução podem variar entre execuções consecutivas (+/- código para executar)

Execução Periódica

— **Características**

— Execuções consecutivas estão separadas por um intervalo de tempo fixo (ie., o período)

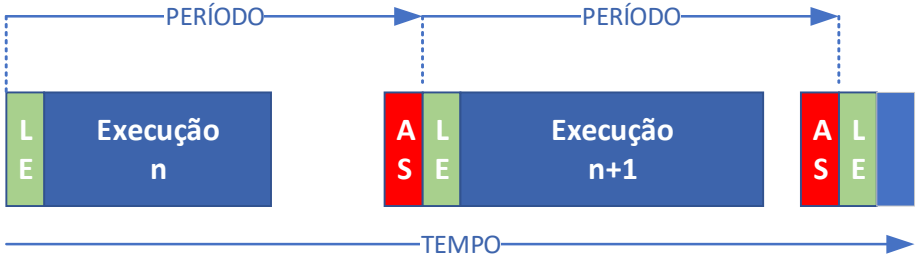




— **Vantagens**

— As entradas são amostradas com um período constante.

— Pode-se estender o mesmo conceito às saídas

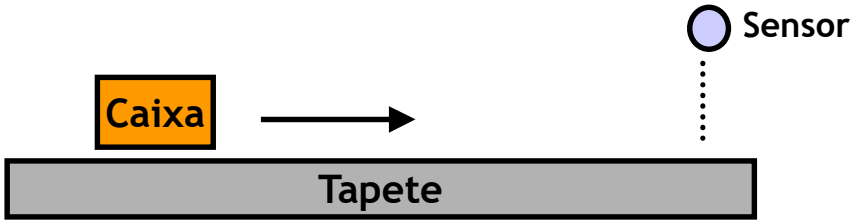


— **Desvantagens**

— Implica utilizar um temporizador para garantir a execução periódica

Execução funcional vs. temporal

Problema: uma caixa é transportada pelo tapete T. Quando o sensor S for ativado, o tapete deve parar em < 20ms, caso contrário a caixa "cai" para fora do tapete.



Programa:

```
IF S THEN
        T := False;
ELSE
        T := True;
END_IF;
```

O algoritmo de controlo está **funcionalmente correto**.

No entanto, se a desativação do tapete não for efetuada num intervalo de tempo < 20ms após a deteção da caixa pelo sensor, o programa **não cumpre as especificações**.

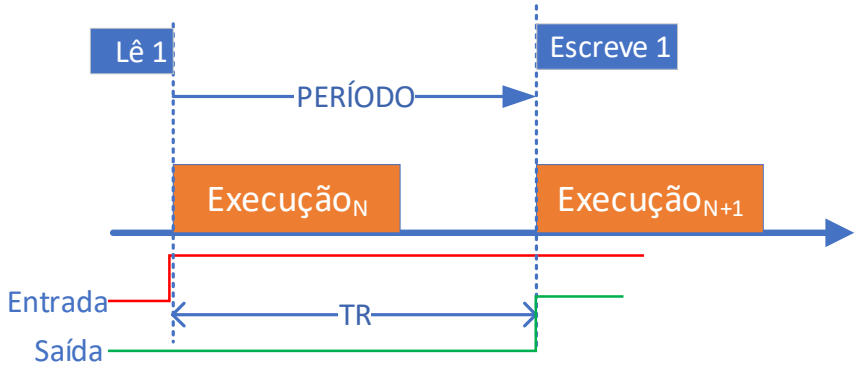
Conclusão: **o algoritmo de controlo está a ser executado de forma incorreta**.

É preciso prever (ou estimar) o tempo de resposta |

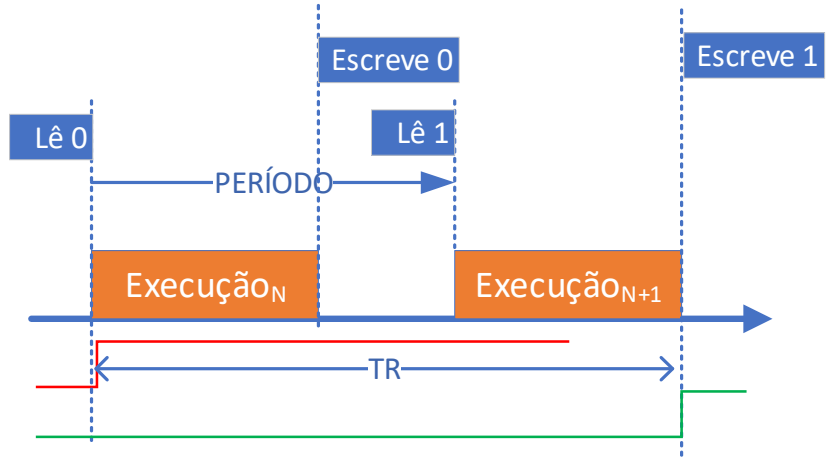
Cálculo do tempo de resposta

Programa: Saída:=Entrada

Execução Periódica

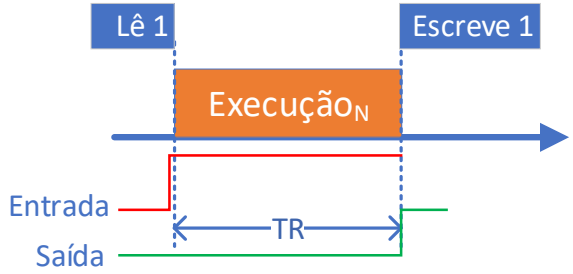


Melhor caso = 1 ciclo

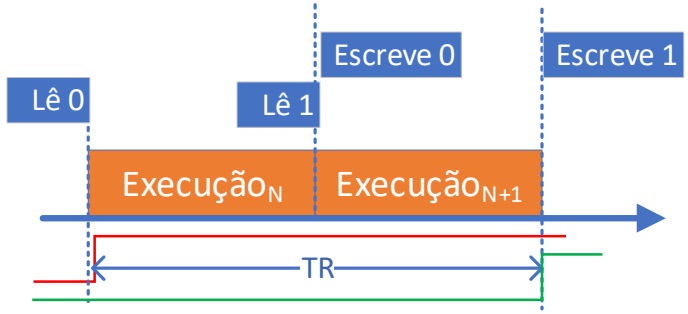


Pior caso = 2 ciclos

Execução Cíclica



Melhor caso = 1 execução



Pior caso = 2 execuções

Porquê adotar estes modelos?

- O facto de todas entradas (sensores) serem adquiridas no mesmo instante e com um intervalo de tempo fixo (ou próximo disso) facilita o projeto de qualquer controlador.
  - Recordar os requisitos da **Teoria da Amostragem** relativos ao período mínimo de aquisição de um sinal que permite a sua reconstrução (i.e. a sua interpretação correta).

- O facto das saídas (atuadores) serem atualizadas no mesmo instante
  - Garante um a alteração síncrona das entradas do processo.
  - Permite evitar situações em que saídas que são mutuamente exclusivas (ex. semáforo) possam ter esse princípio violado.

- Permite, de forma simples, fazer uma previsão fidedigna do **tempo de resposta do sistema**: i.e. quando tempo decorre entre a leitura de uma entrada a atualização de uma saída. Este aspeto é muito importante em sistemas de controlo (i.e. conhecer a 'rapidez' com que o sistema consegue reagir a uma mudança)

Implementação – Execução cíclica

- As definições dos estados e do estado atual têm que ser **globais**

- A utilização de um ciclo **while(1)** para implementar a execução cíclica

- A função **read_inputs()** é responsável pela leitura das entradas (do controlador).

- A função **write_outputs()** é responsável pela escrita das saídas (do controlador)

```
// Estados da máquina
typedef enum{
     stateA,
     stateB,
     ....
     stateZ
} stateNames;


// Estado da máquina
stateNames currentState = stateA;



int main() {

     // Implementação cíclica
     while(1) {

          // Leitura das entradas
          read_inputs();


          // Codigo da ME
          ...

          // Escrita das saídas
          write_outputs();
     }

     return 0;
}
```

Implementação – Execução periódica

- Definição do **tempo de ciclo** (i.e. período de execução) na variável **scan_time.**

- A utilização de um ciclo **while(1)** para implementar a execução periódica.

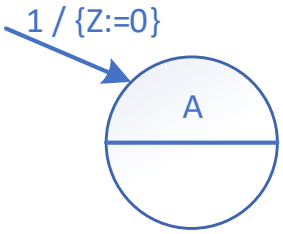
- Utilização da função **sleep_abs()** para a implementação do **período** entre execuções
  - A execução do programa é suspensa durante esse período de tempo.
  - Implementada numa biblioteca à parte.

- Para atualizar as saídas periodicamente basta colocar a função **sleep_abs()** antes de **write_outputs()**

```
// Igual à implementação cíclica
...

// Tempo de ciclo : vamos considerar milissegundos
uint64_t scan_time = 1000;     // 1 segundo
uint64_t scan_time = 10;       // 10 milissegundos



int main() {

    //Implementação periódica
    while(1) {

        // Leitura das entradas
        read_inputs();


        // Codigo da ME
        ...

         // Escrita das saídas
        write_outputs();


        // Aguarda pelo próximo ciclo
        sleep_abs(scan_time);
    }

    return 0;
}

// uint64_t = unsigned int 64 bits
```

Inicialização da ME

— A inicialização da ME pode ser feita utilizando uma função **init_SM()** e invocando-a uma única vez **antes do início do ciclo**.




```
void init_SM()
{
    // Inicialização da ME
    Z=0;
    ...
}


...


int main() {

    // Inicializa a ME
    init_SM();


    while(1) {
        ...
        // Codigo da ME
        ...
    }
}
```

Implementação de Flancos

— A utilização de um modelo cíclico / periódico permite a uma deteção simples de flancos.

— **Abordagem**: Comparar o valor da variável obtido no ciclo anterior com valor atual.

Para cada variável que se pretende detetar um flanco, criar uma variável adicional que tenha o valor dessa variável obtida no ciclo anterior.

Flancos ascendentes: **0→1**

— Valor no ciclo anterior: 0

— Valor no ciclo atual: 1

Flancos descendentes: **1→0**

— Valor no ciclo anterior: 1

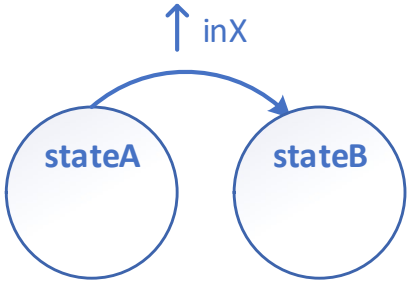
— Valor no ciclo atual: 0

Memorizar a ocorrência do flancos (ascendentes / descendentes) **em outras variáveis adicionais**, para que possam ser utilizadas posteriormente na ME.

Para simplificar este processo, vamos incluir todo este processamento numa função adicional: **edge_detection()** que deve ser colocada depois de **read_inputs()**

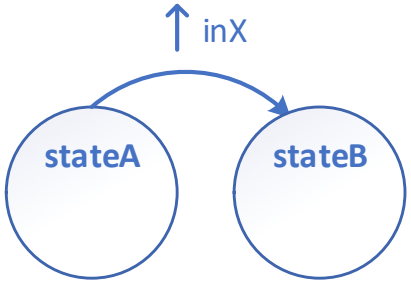
— Necessário para garantir o acesso aos últimos valores lidos

Implementação de Flancos - Exemplos



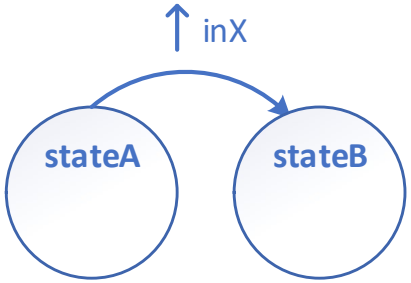

```
#include <stdbool.h> // declarar o tipo Bool


// Estados da máquina
typedef enum{
    None,    // necessário para detetar ativação no 1º ciclo
    stateA,
    stateB
} stateNames;

bool inX;    // Exemplo de uma variável



// Variáveis auxiliares para guardar valores do ciclo anterior
// p -> previous

// Para o estado basta 1 variável
stateNames p_state = None;

// Para as restantes variáveis, é necessária uma por variável
bool p_inX = 0;



// Variáveis auxiliares para guardar flancos
// re -> rising edge
// fe -> falling edge

bool re_stateA = False, fe_stateA = False;
bool re_inX = False, fe_inX = False;
```

Implementação de Flancos - Exemplos



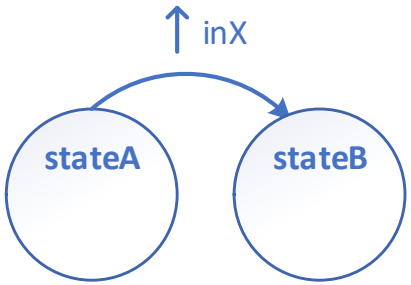

```
// Deteta flancos
void edge_detection() {

    // Detetar flancos de variáveis booleanas


    // Flancos ascendentes
    if (p_inX == False && inX == true)
        re_inX = true;
    else
        re_inX = false;


    // Flancos descendentes
    if (p_inX == true && inX == false)
        fe_inX = true;
    else
        fe_inX = false;


    // No próximo ciclo, o valor anterior passa a ser o atual
    p_inX = inX;
```

Implementação de Flancos - Exemplos




```
            ...

            // Detetar flancos de Estados


            // Flancos ascendentes => ativação de estados
            if (p_state != stateA && currentState == stateA)
                        re_stateA = true;
            else
                        re_ stateA = false;


            // Flancos descendentes => desativação de estados
            if (p_state == stateA && currentState != stateA)
                        fe_stateA = true;
            else
                        fe_ stateA = false;


            // No próximo ciclo o valor anterior passa a ser o atual
            p_state = currentState;


} // edge_detection
```

Implementação de Flancos - Utilização




```
...

int main() {

    while(1) {

        // Leitura das entradas
        read_inputs();

        // Deteta flancos
        edge_detection();

        ...

        // Codigo da ME
        ...

        // Utilização de flancos

        if (re_stateA)
            ...

        if (fe_inX)
            ...

        // Escrita das saídas
        write_outputs();

    }

    return 0;
}
```

Implementação de Temporizadores

- A implementação de temporizadores pode ser realizada:
  - Utilizando uma estrutura de dados **timerBlock** para armazenar informação sobre o temporizador
  - A estrutura **timeBlock** tem 2 campos:

    on : utilizado para iniciar (**True**) / parar (**False**) o temporizador

    time : utilizado para guardar o valor corrente do temporizador

  - Implementando uma função auxiliar **update_timers()** para atualizar os temporizadores em cada ciclo

  - Implementar 2 funções auxiliares, **start_timer()** e **stop_timer()**, para iniciar e parar o temporizador.

```
// Define um tipo para o temporizador

typedef struct {
    bool on;
    uint64_t time;
} timerBlock;


// Declara temporizadores

timerBlock timer1, timer2,...;




// Atualiza temporizadores

void update_timers() {

    // Atualiza temporizadores
    ...
}


// Inicia temporizador
start_timer(&timer1);

...

// Para temporizador
stop_timer(&timer2)
```

Implementação de Temporizadores – Execução periódica

- Definir o tempo de ciclo: **scan_time**
- Declarar as variáveis necessárias aos temporizadores
- A função **update_timers()** deve ser a primeira a ser invocada a seguir ao inicio do ciclo.
- **Utilizar** o temporizador onde necessário
- **Nunca utilizar** expressões do tipo:

  timer.time == valor

  porque sendo o valor da contagem não preciso, esta condição pode nunca ser verdadeira. Utilizar sempre o operador **>=**

```
// Tempo de ciclo
uint64_t scan_time = 100;

// Declara temporizadores
timerBlock timer1, timer2;


int main() {

    while(1) {

        // Atualiza temporizadores
        update_timers();

        ...
        // Codigo da ME
        ...

        // Utiliza o temporizador timer1
        start_timer(&timer1);
        ...
        if (timer1.time >= 1000)
        ...
        stop_timer(&timer1);


        // Aguarda pelo próximo ciclo
        sleep_abs(scan_time);
    }

    return 0;
}
```

Implementação de Temporizadores – Execução periódica

— A função **update_timers()** incrementa apenas os temporizadores que estão ativos com o valor do **tempo de ciclo – que neste caso é constante.**

```
void update_timers() {

     // Atualiza temporizadores

     if (timer1.on)
          timer1.time = timer1.time + scan_time;

     if (timer2.on)
          timer2.time = timer2.time + scan_time;

}
```

Implementação de Temporizadores – Execução cíclica

- Nesta caso é necessário calcular o valor **exato do tempo de ciclo** (porque não é conhecido à partida).
- A estrutura do código da ME é o mesmo !
- A função **update_timers()** incrementa apenas os temporizadores que estão ativos em função do **tempo de ciclo** calculado
- A função **get_time()** permite obter o **tempo absoluto:** relógio do controlador.
  - Implementada numa biblioteca à parte
- As variáveis utilizadas pela função **get_time()** têm que ser **globais.**

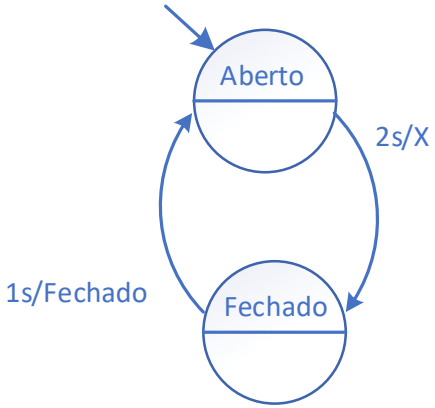
```
// Declara variáveis para calculo do tempo de ciclo
uint64_t start_time=0, end_time=0, cycle_time=0;


// Declara temporizadores
timerBlock timer1, timer2;
...


void update_timers() {

    // Calcula o tempo de ciclo

    end_time = get_time();

    if (start_time == 0)
        cycle_time = 0;
    else
        cycle_time = end_time – start_time;

    // o fim do ciclo atual é o inicio do próximo
    start_time = end_time;


    // Atualiza temporizadores

    if (timer1.on)
        timer1.time = timer1.time + cycle_time;

    if (timer2.on)
        timer2.time = timer2.time + cycle_time;


}
```

Implementação de Temporizadores – Iniciar / Parar

- Para iniciar/parar um temporizador são utilizadas 2 funções adicionais:
  - **start_timer()** : inicia a contagem de tempo de um temporizador
  - **stop_timer()** : pára a contagem de tempo de um temporizador
    - Ambas as funções tem como parâmetro um apontador para o temporizador
- Estas funções **são invocadas em diferentes locais do código da ME**, em função do tipo de temporizador:
  - Se o temporizador depender da ativação / desativação de um estado, devem ser implementadas como **ações associadas a transições**
    - Se o temporizador for ativado no estado inicial deve ser iniciado na função **init_SM()**
  - Se o temporizador depender de uma variável que não está associada a um estado (interna ou externa), devem implementadas como atualizações de saídas:
    - Como se fosse uma atualização de uma variável interna (que neste caso é um temporizador).

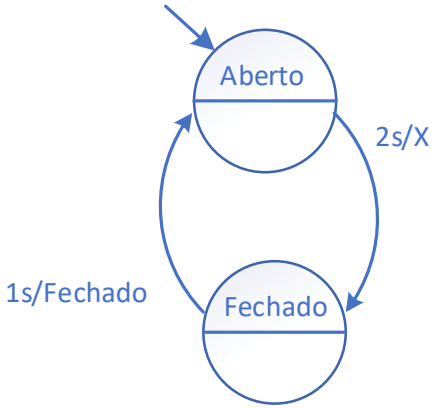
```
void start_timer(timerBlock* t) {
    t->on = true;
    t->time = 0;
}

void stop_timer(timerBlock* t) {
    t->on = false;
    t->time = 0;

    // t->time pode ser comentado
    // se for importante guardar
    // o valor do temporizador
}
```

Implementação de Temporizadores – Iniciar / Parar



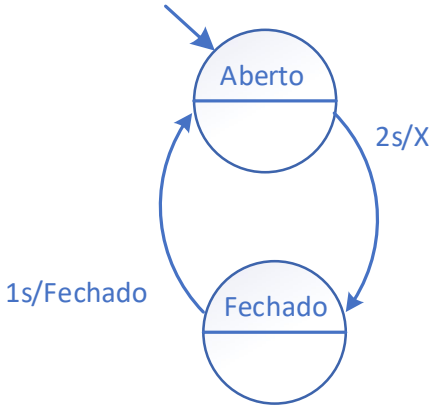

```
// Declara temporizadores
timerBlock timer1;    // temporizador de X
timerBlock timer2;    // temporizador de Fechado
...
// Variáveis auxiliares para detetar flancos
bool p_X = false;
bool re_X = false, fe_X = false;


int main() {

    while(1) {

        // Atualiza temporizadores
        update_timers()

        // Leitura das entradas
        read_inputs();

        // Deteta flancos
        edge_detection();
```

Implementação de Temporizadores – Iniciar / Parar




```
// Transição entre estados
switch (currentState) {

    // Aberto
    case Aberto :
        if (timer1.time > 2000) {

            currentState = Fechado;

            // Inicia temporizador do estado Fechado
            start_timer(&timer2);
        }
    break;

    // Fechado
    case Fechado :
        if (timer2.time > 1000) {

            currentState = Aberto;

            // Pára temporizador do estado Fechado
            stop_timer(&timer2);
        }
    break;


} //end_switch
```

Implementação de Temporizadores – Iniciar / Parar




```
        // Atualização de saídas
        ...

        // Inicia temporizador de X
        if (re_X)
            start_timer(&timer1);

        // Pára temporizador de X
        if (fe_X)
            stop_timer(&timer1);


        // Escrita das saídas
        write_outputs();

        ...

 } // end_while

 return 0;

} // end_main
```

Múltiplas máquinas de estado

- Criar para cada ME uma função que encapsule o seu funcionamento completo: **MEi()**
- Cada ME deve ser invocada dentro do ciclo.
- Devem ser declaradas como **variáveis globais**:
  - A variável de estado de cada ME
  - Todas as variáveis internas da ME
  - Temporizadores
  - Variáveis auxiliares de flancos

```
void ME1() {
      switch (currentState1) {
            ...
      }
}

void M2() {
      switch (currentState2) {
            ...
      }
}
```

```
// Estados das ME
typedef enum{...} stateNames1;
typedef enum{...} stateNames2;
...

// Variáveis Globais
stateNames1 currentState1;
stateNames2 currentState2;
...

int main() {
            // Inicializa ME1
            ...
            // Inicializa ME2
            ...
            while(1) {
                        // Atualiza temporizadores
                        ...
                        // Leitura das entradas
                        ...
                        // Deteta flancos
                        ...
                        // Executa ME1
                        ME1();

                        // Executa ME2
                        ME2();
                        ...
                        // Escrita das saídas
            }
}
```

Múltiplas máquinas de estados

— Quando se tem múltiplas máquinas de estados e estas têm transições que dependem o estado de outras máquinas, a **ordem pela qual as máquinas são executadas torna-se relevante** e se não forem tomados cuidados podem existir problemas inesperados.

— Exemplo

— A ordem pela qual **M1** e **M2** executadas produz resultados diferentes:

  — M1 → M2: **OUT1** é ativada e **OUT2** não é ativada

  — M2 → M1: **OUT2** é ativada e **OUT1** não é ativada

— Para que não existam problemas com a **ordem** pela qual as ME são executadas, devemos adotar um **modelo síncrono** de atualização dos estados, tal como já é feito para as entradas e saídas:

  — Ou seja, o próximo estado das máquinas só é atualizado no **fim da execução de todas as máquinas**.

  — Isto permite que todas as máquinas tenham a mesma imagem do estado do sistema no inicio da sua execução.

Múltiplas máquinas de estado

— A forma mais simples de implementar este modelo **é não atualizar de imediato o estado da máquina**, mas sim guardar essa informação numa variável intermédia: **nextState**.

— No final da execução de todas as máquinas, utilizar essa variável para atualizar o estado da máquina.

```
// Variáveis Globais
stateNames1 currentState1; // Estado atual
stateNames2 currentState2;

stateNames1 nextState1; // Próximo estado
stateNames2 nextState2;

...

     while(1) {
          ...
          // Executa ME1
          ME1();

          // Executa ME2
          ME2();
          ...

          // Atualiza estados
          currentState1 = nextState1;
          currentState2 = nextState2;

          // Escrita das saídas
          ...

     }
```

```
void ME1() {

     switch (currentState1) {

          case A :
               if (currentState2 == C)
                    nextState1 = B;
          break;
          ...
     }
}


void M2() {

     switch (currentState2) {

          case C :
               if currentState1 == A)
                    nextState2 = D;
          break;
          ...
     }
}
```

Interligação ao processo : sensores e atuadores

- A interligação aos sensores e atuadores está dependente da plataforma computacional utilizada.
- Para sistematizar o procedimento e torna-lo independente da plataforma, vamos admitir que é disponibilizada uma biblioteca **IO.h (*.c)** onde estão:
  - Declaradas todas as variáveis associadas aos **sensores** e **atuadores** (variáveis globais)
  - A implementação das funções:
    - **read_inputs()**
    - **write_outputs()**
    - **sleep_abs()**
    - **get_time()**
- A inclusão desta biblioteca realiza-se logo no inicio do programa que implementa a ME

```
// IO.h
#include <stdbool.h>

void read_inputs();
void write_outputs();
void sleep_abs();
uint64_t get_time();


// Variáveis globais associadas
// a sensores / atuadores
...
```

```
----------------------------------------
-

#include "IO.h"
...

int main() {

            ...
            // Codigo da ME
            ...
}
```

Debug

- É importante incluir na implementação da ME código que permita fazer degug (depuração) da mesma. Existem muitas alternativas.
- A solução proposta passa por utilizar instruções condicionais de **pré processamento** que permitem de forma ágil ativar / desativar código utilizado para fazer debug, bastando para isso:
  - #define **XXXX** para ativar a diretiva
  - #undef **XXX** para desativar a diretiva
- Podem der criadas múltiplos tipos de debug

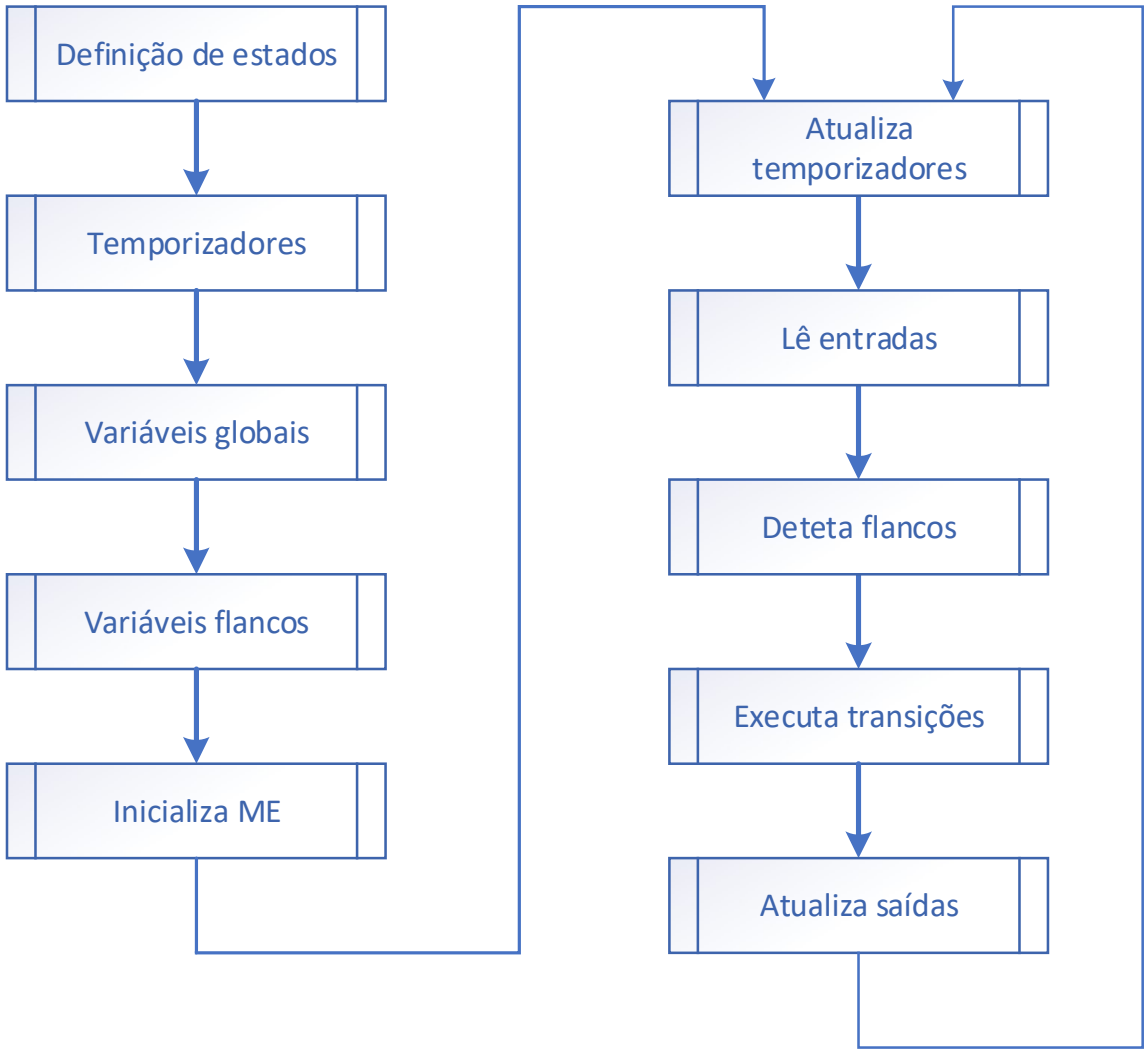
```
#define DEBUG
#undef DEBUG_ESTADOS


// Necessário fazer debug aqui
#ifdef DEBUG
printf (“debug...\n”);
#endif


// Necessário fazer debug dos estados aqui
#ifdef DEBUG_ESTADOS
printf (“debug estados... \n”);
#endif
```

Debug - Estados

— Uma forma simples de fazer o debug dos estados é criar um *array* de *strings*, com o nome dos estados, e utilizar o estado corrente como *index* do arra.y.

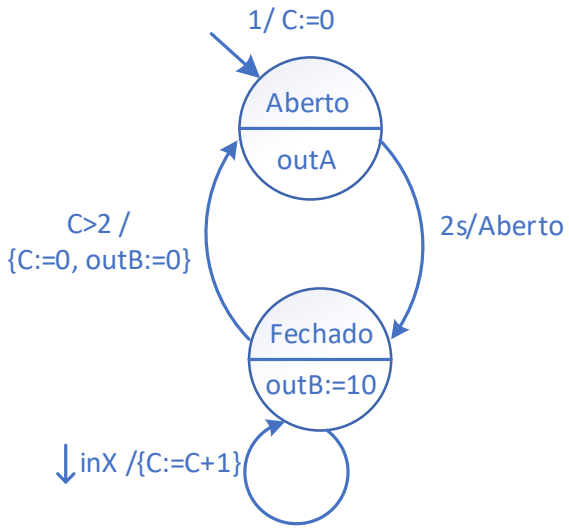
```
#define DEBUG_ESTADOS



// Estados da máquina
typedef enum{
    Aberto = 0, // necessário para o index 0 do array
    Fechado
} stateNames;


// Strings debug
char * stateNamestrings[] = {
          “Aberto",
          “Fechado",
};  // Nomes dos estados

// Estado atual da máquina
stateNames currentState = Aberto;


// Imprime estado atual da máquina
#ifdef DEBUG_ESTADOS
printf(“Estado =%s\n",stateNamesStrings[currentState]);
#endif
```

Fluxograma de implementação final



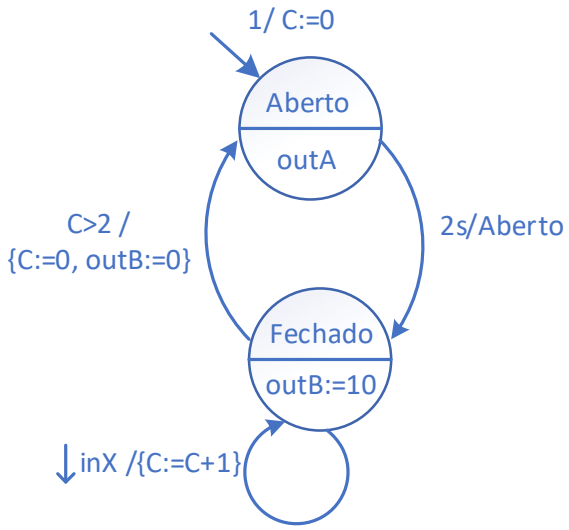
EXEMPLO COMPLETO

Exemplo completo:



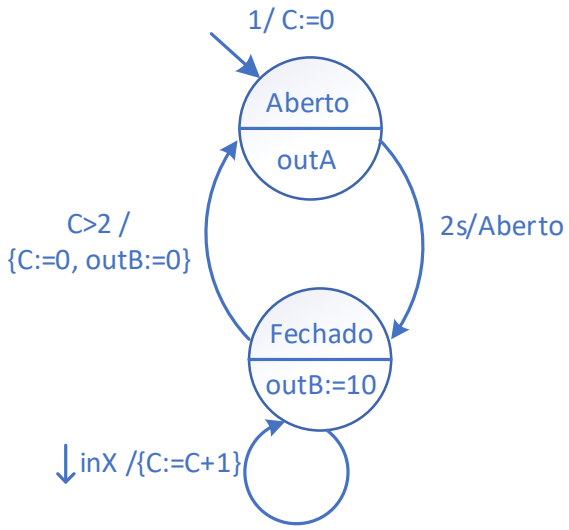

```
// ---------- IO.h -------
#include <stdbool.h>

// Funções
void read_inputs();
void write_outputs();
void sleep_abs();
uint64_t get_time();

// Sensores
bool inX = 0;

// Atuadores
bool outA = 0;
uint8_t outB = 0;



void read_inputs() {
     ...
     inX = ...;
     ...
}

void write_outputs() {
     ...
     ... = outA;
     ...
}
```

Exemplo completo:



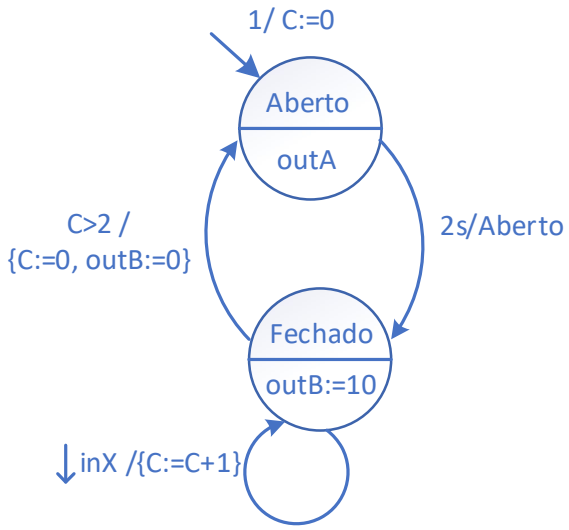

```
// ---------- Máquina de estados -----------

#include “IO.h”
#include <stdio.h>

#define DEBUG

// Tipos de dados

// Temporização
typedef struct {
    bool on;
    uint64_t time;
} timerBlock;


// Estados da máquina
typedef enum{
    Aberto,
    Fechado
} stateNames;


// Funções
void edge_detection();
void update_timers();
void start_timer(timerBlock* t);
void stop_timer(timerBlock* t);
void init_SM();
```

Exemplo completo:



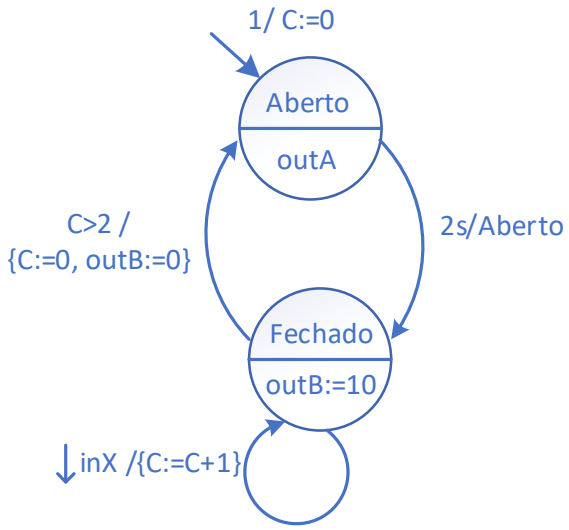

```
// Estado atual da máquina
stateNames currentState = Aberto;

// Temporizadores
timerBlock timer1;    // temporizador do estado Aberto

// Variáveis globais
uint8_t C;

// Variáveis auxiliares para detetar flancos
bool p_inX=false, fe_inX= false;


// Tempo de ciclo
uint64_t scan_time = 1000;     // 1 segundo


// Inicializa a ME
void init_SM()
{
    // Inicialização
    C=0;

    // Inicia temporizador do estado Aberto
    start_timer(&timer1);
}
```

Exemplo completo:



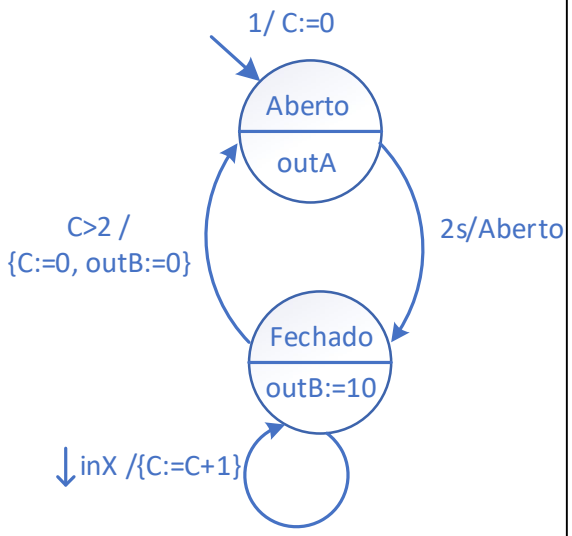

```
// Código principal
int main() {

    // Inicialização da ME
    init_SM();

    // Ciclo de execução
    while(1) {

        #ifdef DEBUG
        printf ("-- Inicio do Ciclo --\n");
        #endif

        // Atualiza temporizadores
        update_timers();


        // Leitura das entradas
        read_inputs();


        // Deteta flancos
        edge_detection();
```

Exemplo completo:



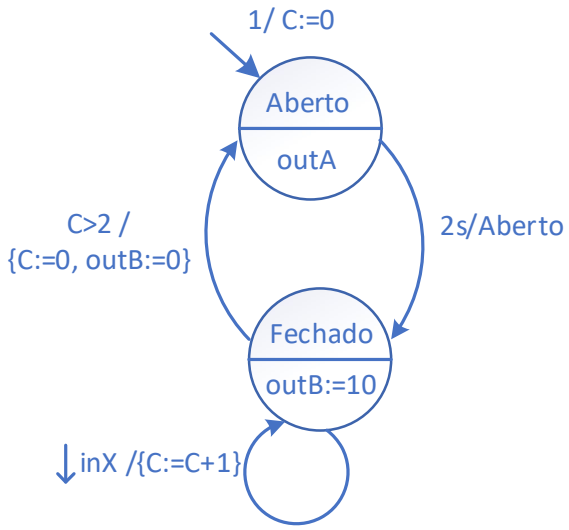

```
// Transição entre estados
switch (currentState) {

    case Aberto :

        if (timer1.time >= 2000){

            currentState = Fechado;

            stop_timer(&timer1);
        }

    break;

    case Fechado :
        if (fe_inX)
            C = C+1;

        if (C > 2) {

            currentState = Aberto;

            C = 0;
            outB = 0;
            start_timer(&timer1);
        }
    break;

} //end switch
```

Exemplo completo:



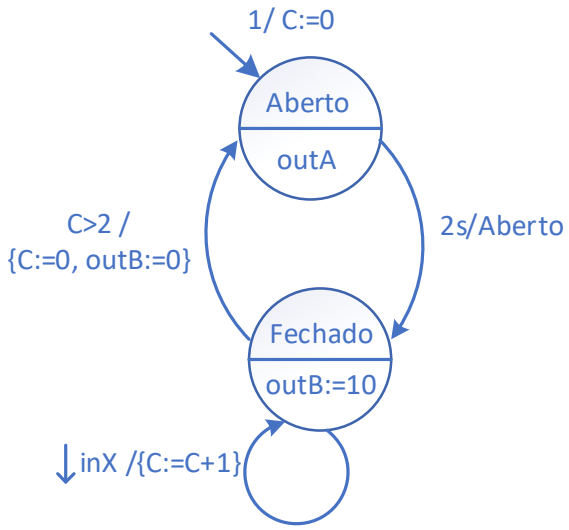

```
        // Atualiza saídas

        // Saídas booleanas
        outA = (currentState == Aberto);


        //Saídas não booleanas
        if (currentState == Fechado)
            outB = 10;


        //Escrita nas saídas
        write_outputs();


        //Aguarda pelo próximo ciclo
        sleep(scan_time);


    } // end while

    return 0;

} // end main
```

Exemplo completo:




```
void edge_detection() {

            // Variável X, FE

            if (p_inX && !inX)
                        fe_inX = true;
            else
                        fe_inX = false;

            p_inX = inX; // Guarda novo valor


}
```

Exemplo completo:




```
void update_timers() {

            // Atualiza temporizador
            if (timer1.on)
                        timer1.time = timer1.time + scan_time;
            }


void start_timer(timerBlock* t) {
            t->on = true;
            t->time = 0;
}


void stop_timer(timerBlock* t) {
            t->on = false;
            t->time = 0;
}
```