## CHƯƠNG 3. XÂY DỰNG CHƯƠNG TRÌNH PHẦN MỀM

### 1. CẤU TRÚC CHƯƠNG TRÌNH PHẦN MỀM.

| Khởi tạo |
|---|
| 1. Nạp từ bộ nhớ chương trình<br>2. Xóa và cài đặt các thông số của thanh ghi<br>3. Kiểm tra bộ nhớ chương trình, nếu có lỗi chuyển đên chưương trình xử lý lỗi (ercode=3)<br>4. Kiểm tra ALU, MAC, ADC, DAC nếu có lỗi thông báo lỗi (ercode=4)<br>5. Kiểm tra truyền thông, nếu có lỗi thông báo lỗi (ercode=5)<br>6. Gọi chương trình khởi tạo phần mềm<br>7. Chuyển đến mục 8 |
| Hàm ngắt |
| 8. Xác định giá trị phản hồi, nếu có lỗi thông báo lỗi ercode=8<br>9. Đọc giá trị đặt đầu vào<br>10. Tính toán giá trị tác động điều khiển<br>11. Đưa ra giá trị ra<br>12. Chuyển đến mục 13 |
| Hàm thực hiện chương trình |
| 13. Khổi tạo Watchdog Timer<br>14. Xây dựng các thuất toán có mức độ ưu tiên thấp<br>15. Xóa giá trị của Watchdog, nếu xuất hiện tràn thời gian của watchdog thông báo lỗi ercode=15<br>16. Chuyển đến mục 14 |
| Khởi tạo phần mềm |
| I1. Kiểm tra trạng thái của đối tượng điều khiển nếu có lỗi thông báo lỗi ercode I1<br>I2. Kiểm tra và cài đặt thông số ban đầu biến bộ nhớ<br>I3. Trở về |
| Hàm lỗi |
| E1. Kiểm tra xác định mã lỗi, cố gắng khối phục lỗi, dừng sưk làm việc của cơ cấu chấp hành |

| Hàm ngắt do Watchdog |
| --- |
| W 1. Chuyển đến thông báo lỗi ercode W1 |

Trước khi xây dựng chương trình phần mềm ta quan tâm đến một số đặc điểm khi chế tạo mạch phần cứng tối thiêu cho vi xử lý hoạt động gồm:

a. Mạch tạo dao động:

Tất cả các loại vi xử lý đều được điều khiển bởi 1 vài dạng mạch tạo dao động.

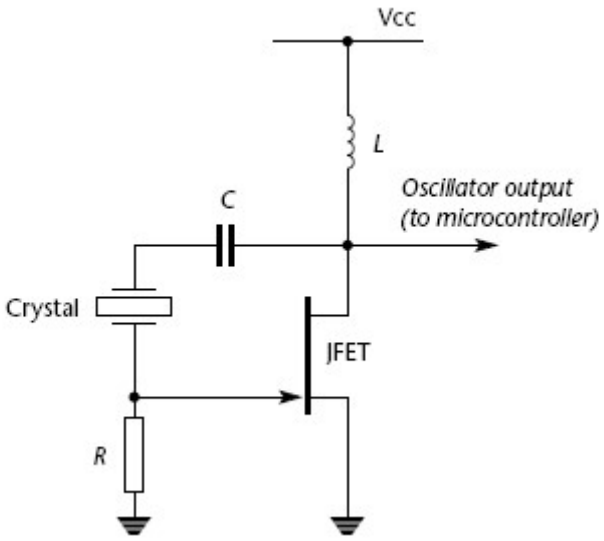
Mạch tạo dao động giống như nhịp tim của hệ thống, có tính chất quan trọng trong sự làm việc của hệ thống vi xử lý.

nếu mạch tạo dao động bị lỗi, hệ thông không thể làm việc, nếu mạch tạo dao động hoạt động có tần số không ổn định dẫn đến việc tính toán thời gian của hệ thống bị sai.

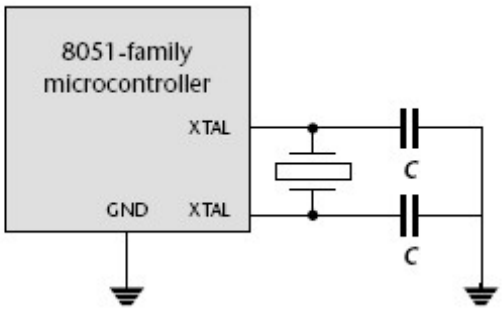
Tính ổn định của dao động thể hiện bằng thông số pmm, ví dụ ±20ppm có nghĩa là trong số 1 triệu xung của dao động sẽ sai lệch ±20 xung (nếu sử dụng dao động này làm xung nhịp đồng hồ thì tương ứng với 1 năm sẽ gây ra sai lệch đồng hồ là 10 phút).

Với bộ dao động sử dụng thạch anh thông số này nằm trong khoảng từ ±10 đến ±100ppm (tức là sai số từ 5 đến 50 phút trong 1năm). Với bộ dao động bằng gốm Ceramic sai số khoảng ±5000 ppm tương đương 50 phút mỗi tuần.

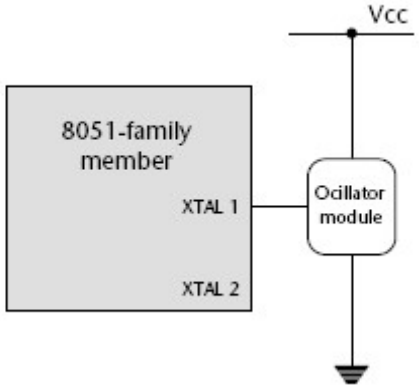
Hiện tại có những bộ tạo dao động có bù nhiệt (TCXO) có thể đạt độ ổn định ±0.1ppm tức đạt sai số 1 phút trong 20 năm.



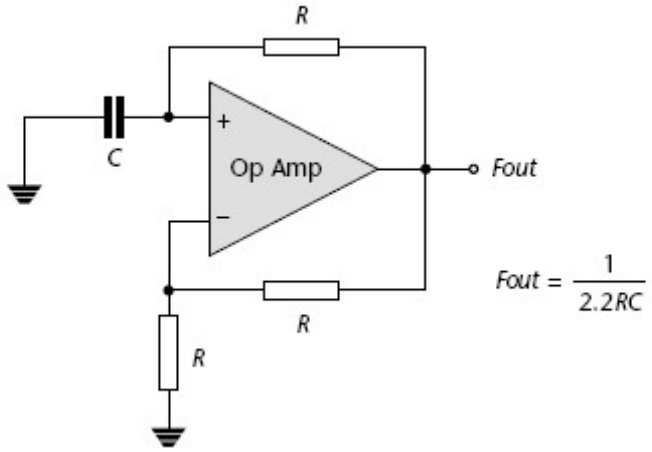

mạch dao động dùng tranzitor JFET



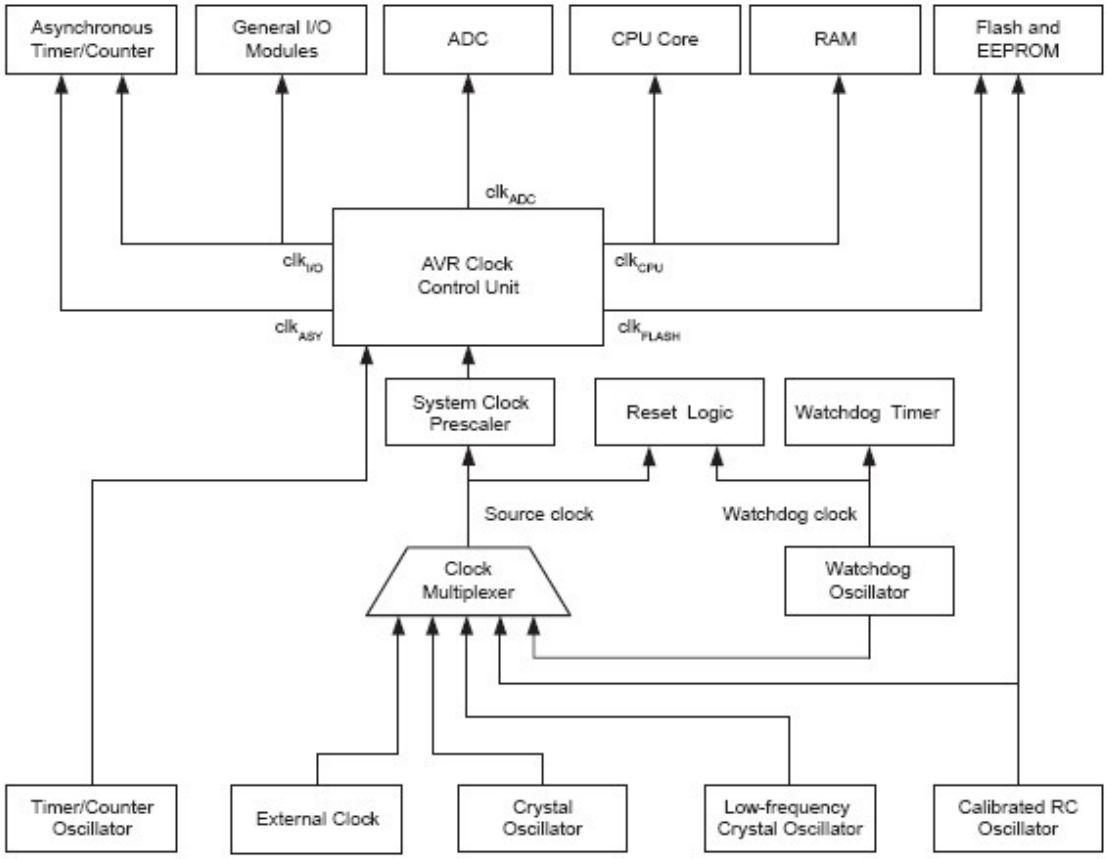

Mạch tạo dao động dùng mạch trong Chip




Mạch tạo dao động dùng Chip ngoài




Mạch tạo dao động RC.



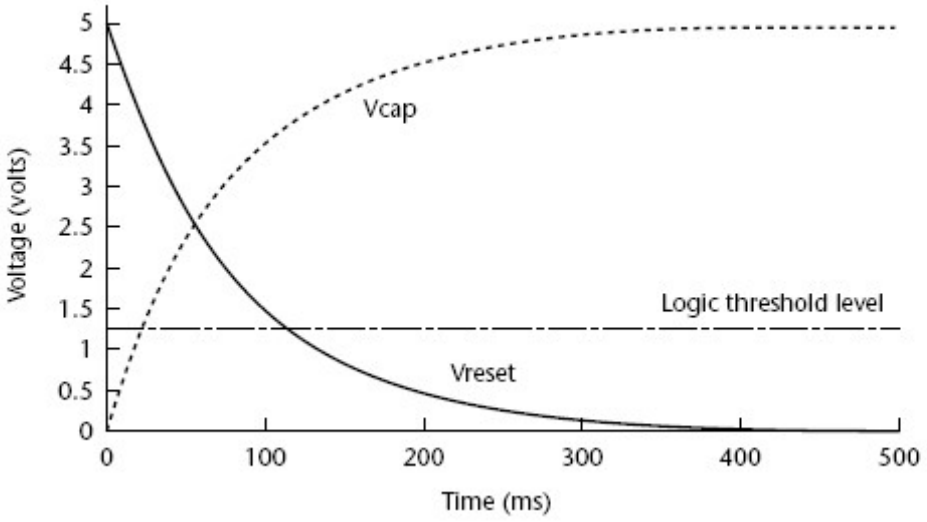

Sơ đồ mạch dao động và bộ chia tần trong Chip vi xử lý

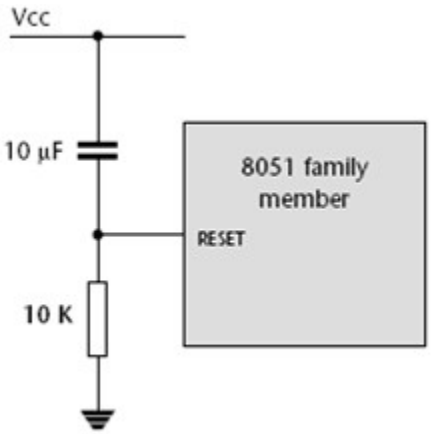
b. Mạch Reset.

Đây là trạng thái bắt buộc khi vi xử lý bắt đầu hoạt động, nó đảm nhiệm vai trò khởi tạo hệ thống. Một xung có giá trị logic phù hợp (mức 0 hay mức 1)

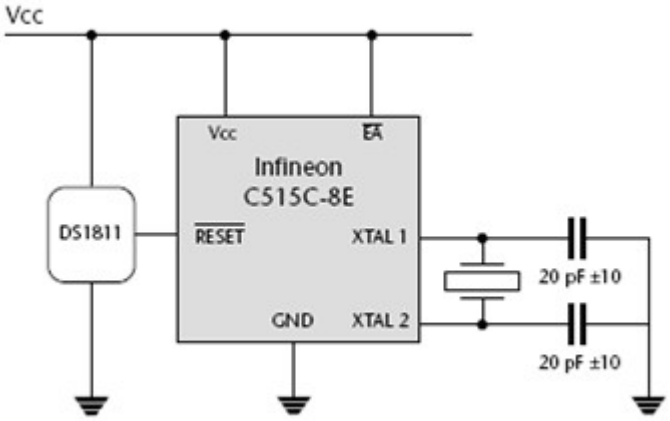
đưa vào chân Reset sẽ chuyển vi xử lý về trạng thái bắt đầu hoạt động (con trỏ chương trình về giá trị 0x00, các giá trị thanh ghi và  bộ nhớ bị xóa).
Quá trình Reset cần phải tiến hành trong khoảng thời gian nhất định và hòi hỏi mạch tạo dao động phải làm việc. các mạch Reset có thể đa dạng, đơn giản nhất là dùng mạch tạo xung sử dụng mạch RC

Sự khác nhau thể hiện ở độ dốc và thời gian tồn tại xung Reset.




Mạch Reset dùng RC




Mạch dùng IC tạo xung Reset

## II. XÂY DỰNG HỆ ĐIỀU HÀNH NHÚNG

1. Vòng lặp chính.

Một chương trình cho hệ nhúng đơn giản nhất gồm một vòng lặp vô tận (vòng quét chính) có cấu trúc như sau:

```
/*------------------------------------------------------------------ *-

   Main.C

  ------------------------------------------------------------------

   Architecture of a simple Super Loop application

   [Compiles and runs but does nothing useful]

-*------------------------------------------------------------------*/

#include "X.h"

/*------------------------------------------------------------------*/
void main(void)
   {
   // Prepare for task X
   X_Init();
   while(1) // 'for ever' (Super Loop)


       {
       X();  // Perform the task
       }
    }

  /* ------------------------------------------------------------------*-
     -------END OF FILE ------------------------------------------
  -*------------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------------*-
   X.H
  ------------------------------------------------------------------
   - see X.C for details.
-*-------------------------------------------------------------------*/
// Function prototypes
void X_Init(void);
void X(void);
/*-------------------------------------------------------------------*-
  ----- END OF FILE ----------------------------------------------
   -*-------------------------------------------------------------------*/

     /*-------------------------------------------------------------------*-
        X.C
       ------------------------------------------------------------------
        'Task' for a demonstration Super Loop application
        [Compiles and runs but does nothing useful]
     -*-------------------------------------------------------------------*/
     /*-------------------------------------------------------------------*/
     void X_Init(void)
        {
        // Prepare for task X
        // User code here ...
        }
     /*-------------------------------------------------------------------*/

  void X(void)
     {
     // Perform task X
     // User code here ...
     }
  /*-------------------------------------------------------------------*-
    -----END OF FILE ----------------------------------------------
  -*-------------------------------------------------------------------*/
```

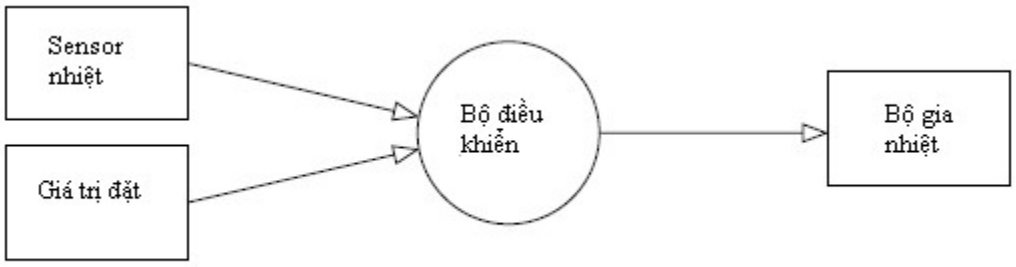
Chương trình gồm 3 file: file main.c, x.h và h.c

Với mục tiêu thực hiện nhiệm vụ X (hàm void x (void)), trước đó cần khởi tạo các giá trị ban đầu (hàm void X_init(void)). Nhiệm vụ X sẽ được thực hiện đên khi ngắt nguồn ra khỏi thiết bị.

Ta thấy vòng lặp while (1) không đòi hỏi nguồn tài nguyên phần cứng đặc biệt như bộ nhớ dữ liệu, bộ định thời. Nó chỉ đòi hỏi một vài byte bộ nhớ chương trình, các ưu điểm chính của chương trình thực hiện theo vòng quét chính là:

- Đơn giản, dễ hiểu
- Không đòi hỏi bộ nhớ dữ liệu hoặ nguồn tài nguyên Chip khi xây dựng chương trình.

Ví dụ một chương trình thực hiện vòng quét chính: Điều khiển nhiệt độ lò nhiệt theo sơ đồ sau:



```
/*-------------------------------------------------------------------*-

   Main.C

  ------------------------------------------------------------------

   Framework for a central heating system using 'Super Loop'.

   [Compiles and runs but does nothing useful]

-*-------------------------------------------------------------------*/

#include "Cen_Heat.h"

/*-------------------------------------------------------------------*/
void main(void)
   {
   // Init the system
   C_HEAT_Init();

   while(1) // 'for ever' (Super Loop)
      {
      // Find out what temperature the user requires
      // (via the user interface)
      C_HEAT_Get_Required_Temperature();

      // Find out what the current room temperature is
      // (via temperature sensor)
      C_HEAT_Get_Actual_Temperature();

      // Adjust the gas burner, as required
      C_HEAT_Control_Boiler();
      }
   }

/*-------------------------------------------------------------------*-
  ----- END OF FILE -------------------------------------------------
-*-------------------------------------------------------------------*/


/*-------------------------------------------------------------------*-

   Cen_Heat.H

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

```
      - see Cen_Heat.C for details.
   -*---------------------------------------------------------------------*/
   // Function prototypes
   void C_HEAT_Init(void);
   void C_HEAT_Get_Required_Temperature(void);
   void C_HEAT_Get_Actual_temperature(void);
   void C_HEAT_Control_Boiler(void);
   /*---------------------------------------------------------------------*-
    ---- END OF FILE ------------------------------------------------
   -*---------------------------------------------------------------------*/
 /*---------------------------------------------------------------------*-
    Cen_Heat.C
 ----------------------------------------------------------------------
    Framework for a central heating system using 'Super Loop'.
    [Compiles and runs but does nothing useful]
 -*---------------------------------------------------------------------*/
 /*---------------------------------------------------------------------*/
 void C_HEAT_Init(void)
    {
    // User code here ...
    }
 /*---------------------------------------------------------------------*/


void C_HEAT_Get_Required_Temperature(void)
   {
   // User code here ...
   }
/*---------------------------------------------------------------------*/
  void C_HEAT_Get_Actual_temperature(void)
     {
     // User code here ...
     }
  /*---------------------------------------------------------------------*/
```

```
void C_HEAT_Control_Boiler(void)
   {
   // User code here ...
   }

/*-------------------------------------------------------------------*-
  -------- END OF FILE ------------------------------------------
-*-------------------------------------------------------------------*/
```

Trong đó :

void C_HEAT_Init(void) : khởi tạo giá trị biến ban đầu

void C_HEAT_Get_Required_Temperature(void): hàm xác định giá trị đặt nhiệt độ

void C_HEAT_Get_Actual_temperature(void): hàm xác định nhiệt độ hiện tại

void C_HEAT_Control_Boiler(void): hàm điều khiển thiết bị gia nhiệt.

Ta thấy trong vòng lặp while(1) các hàm C_HEAT_Get_Required_Temperature, C_HEAT_Get_Actual_temperature, C_HEAT_Control_Boiler được thực hiện một cách tuần tự liên tiếp nhau và việc xác định chính xác chu kỳ thực hiện rất khó khăn.

Trong thực tế có nhiều bài toán cần thực hiện các nhiệm vụ có tính chất chu kỳ theo thời gian định trước ví dụ như:

- Giá trị đo nhiệt độ cần xác định với chu kỳ 0.5 giây
- Màn hình cần làm tươi 40 lần trong 1 giây
- Xác định giá trị đặt với chu kỳ 0.5 giây
- Hàm tính toán giá trị ra thực hiện 20 lần /giây
- Bàn phím cần đọc với chu kỳ 0.2 giây
- Truyền dữ liệu sang thiết bị khác 1 lần/ giây.

Để thực hiện một trong các nhiệm vụ trên, giả sử cần đọc bàn phím với chu lỳ 0.2 giây ta thực hiện chương trình như sau:

```
/*-------------------------------------------------------------------*/
void main(void)
   {
   Init_System();
   while(1) // 'for ever' (Super Loop)
      {
      X();  // Perform the task (10 ms duration)
```

```
    Delay_190ms();  // Delay for 190 ms
    }
  }
```

Trong đó hàm đọc bàn phím (hàm X) thực hiện trong 10ms, ta cần trễ 190ms (hàm Delay_190ms) để sau 200ms hàm X được gọi lại.
Phương pháp khác là sử dụng ngắt thời gian Timer như sau:

```
#include <p24Fxxxx.h>
#define TIMER_PERIOD              20000
#define STOP_TIMER_IN_IDLE_MODE    0x2000
#define TIMER_SOURCE_INTERNAL      0x0000
#define TIMER_ON                  0x8000
#define GATED_TIME_DISABLED       0x0000
#define TIMER_16BIT_MODE          0x0000
#define TIMER_PRESCALER_1         0x0000
#define TIMER_PRESCALER_8         0x0010
#define TIMER_PRESCALER_64        0x0020
#define TIMER_PRESCALER_256       0x0030
#define TIMER_INTERRUPT_PRIORITY   0x1000

int count;
void X(void)
{
  //Insert Code here 10ms
}
void TickInit( void )  //1uS
{
   TMR4 = 0;
   PR4 = TIMER_PERIOD;// 20000
   T4CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
        GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS1bits.T4IF = 0;          //Clear flag
   IEC1bits.T4IE = 0;          //Enable interrupt
   T4CONbits.TON = 0;          //Run timer
```

```
//// Timer 2  10ms
    TMR2 = 0;
   PR2 = 10000;
   T2CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
         GATED_TIME_DISABLED     |     TIMER_16BIT_MODE     |     TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS0bits.T2IF = 0;            //Clear flag
   IEC0bits.T2IE = 1;            //Enable interrupt
   T2CONbits.TON = 1;            //Run timer
}
void __attribute__((interrupt, shadow, auto_psv)) _T2Interrupt(void)  //10ms
{

   // Clear flag
   IFS0bits.T2IF = 0;
   T2CONbits.TON = 0;
if(count++==20)  //200ms
  {
          X();
        count=0;
  }
}

int main(void)
{
        TickInit ();
        while(1)
        {

        }
}
```

Việc thực hiện hàm ngắt Timer cho phép hàm X thực hiện chính xác sau 200ms Thời gian lặp lại việc thực hiện hàm X() được tính như sau:

$$T=T_b+T_i+T_e$$

$T_b$: thời gian thực hiện lệnh gọi hàm ngắt

$T_i$: thời gian bộ đếm Timer bị tràn

$T_e$: thời gian thực hiện lệnh trở về chương trình chính.

Điều kiện để hàm X() được thực hiện chính xác là $T \geq T_x$. Trong đó $T_x$ là thời gian thực hiện hàm X().

Để hệ thống giành khoảng thời gian thực hiện các sự kiện khác là $T >> T_x$

Trong trường hợp cần thực hiện nhiều hàm sự kiện với chu kỳ khác nhau phải thêm biến đếm phụ cho từng sự kiện như sau:

```
#include <p24Fxxxx.h>
#define TIMER_PERIOD            20000
int count_donhiet,count_set,count_kb,count_pid;
void X(void)
{
  // Doan ma xac dinh gia tri do
}
void X1(void)
{
  // Doan ma xac dinh gia tri dat
}
 void X2(void)
{
  // Doan ma doc ban phim
}
 void X3(void)
{
  // Doan ma tinh gia tri PID
}
void TickInit( void )  //1uS
{
   TMR4 = 0;
   PR4 = TIMER_PERIOD;// 20000
   T4CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
        GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS1bits.T4IF = 0;          //Clear flag
   IEC1bits.T4IE = 0;          //Enable interrupt
```

```
   T4CONbits.TON = 0;            //Run timer


//// Timer 2  10ms
    TMR2 = 0;
   PR2 = 10000;
   T2CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
         GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS0bits.T2IF = 0;            //Clear flag
   IEC0bits.T2IE = 1;            //Enable interrupt
   T2CONbits.TON = 1;            //Run timer
}

void __attribute__((interrupt, shadow, auto_psv)) _T2Interrupt(void)  //10ms
{

   // Clear flag
   IFS0bits.T2IF = 0;
   T2CONbits.TON = 0;
 if(count_donhiet++==50)  //500ms
 {
   count_donhiet=0;
         X();      //Do nhiet
         X1();  // Xac dinh gia tri dat
 }
 if(count_kb++==20)  //200ms
 {
   count_kb=0;
         X2();    //Doc pham phim
 }

 if(count_pid++==5)  //50ms
 {
   count_pid=0;
         X3();    //Tinh ham PID
 }
}
```

```
int main(void)
{
        TickInit ();
        while(1)
        {

        }
}
```

## III. Xây dựng chương trình ứng dụng.

### 3.1. Điều khiển đèn giao thông.

Đối với 1 sự kiện trong nhiều trường hợp khu hoạt động có nhiều trạng thái khác nhau. Ví dụ như thực hiện điều khiển đèn tín hiệu giao thông theo 1 hướng trạng thái đèn lần lượt là:

Đèn đỏ sáng 50s
Đèn vàng sáng 3s
Đèn xanh sáng 25s

Như vậy lục đầu có thể hệ thống ở trạng thái A thực hiện hàm Function_A() sau 50 giây chuyển sang trạng thái B thực hiện hàm Function_B() trong 3 giây sau đó chuyển sang trạng thái C thực hiện hàm Function_C() trong 25 giây và chuyển lại trạng thái A.

Các trạng thái của các sự kiện được chia thành các dạng sau:

- Phụ thuộc vào thời gian: Việc chuyển trạng thái chỉ phụ thuộc vào thời gian.
- Phụ thuộc vào đầu vào và thời gian: Đây là dạng phổ biến nhất của hệ thông nhúng, việc chuyển trạng thái phụ thuộc vào cả hai yếu tố là thời gian và trạng thái đầu vào của hệ thống. Ví dụ như hệ thống chỉ chuyển từ trạng thái A sang B khi nhận được 1 giá trị đầu vào trong khoảng x giây thì đầu ra mới tác động.
- Phụ thuộc vào đầu vào: Đây là dạng tương đối thường gặp, việc chuyển trạng thái phụ thuộc vào trạng thái đầu vào của hệ thống. Nếu đầu vào không thay đổi thì hệ thống vẫn ở trạng thái cũ.

Với sự kiện mà trạng thái chỉ phụ thuộc vào thời gian ta xây dựng chương trình với cấu trúc sau:

- Hệ thống sẽ làm việc lớn hơn 2 trạng thái
- Mỗi trạng thái sẽ thực hiện 1 hoặc nhiều hàm
- Việc chuyển trạng thái sẽ được điều khiển bởi sự kiểm soát thời gian.
- Việc chuyển trạng thái có thể bao gồm việc gọi hàm.

```
#include <p24Fxxxx.h>

#define  RED_DURATION 50
#define  GREEN_DURATION 25
#define  AMBER_DURATION 5

#define TIMER_PERIOD            20000
#define ON 1
#define OFF 0

    #define Red_Light                LATDbits.LATD6
    #define Red_Light _TRIS          TRISDbits.TRISD6

    #define Green_Light              LATDbits.LATD5
    #define Green_Light _TRIS        TRISDbits.TRISD5
    #define Amber_Light              LATDbits.LATD4
    #define Amber_Light _TRIS        TRISDbits.TRISD4

typedef enum{RED,GREEN,AMBER}  eLight_State;

eLight_State Light_State_G;
int Timer_In_State,count;
char Call_count_G=0;
void TRAFFIC_LIGHT_Updata(void);
void TickInit( void )  //1uS
{
   TMR4 = 0;
   PR4 = TIMER_PERIOD;// 20000
   T4CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
        GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS1bits.T4IF = 0;          //Clear flag
```

```
   IEC1bits.T4IE = 0;            //Enable interrupt
   T4CONbits.TON = 0;            //Run timer


//// Timer 2  10ms
    TMR2 = 0;
   PR2 = 10000;
   T2CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
         GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS0bits.T2IF = 0;            //Clear flag
   IEC0bits.T2IE = 1;            //Enable interrupt
   T2CONbits.TON = 1;            //Run timer
}

void __attribute__((interrupt, shadow, auto_psv)) _T2Interrupt(void)  //10ms
{

   // Clear flag
   IFS0bits.T2IF = 0;
   T2CONbits.TON = 0;
         count++;
         if(count==100)  //1s
         {
          count=0;
          TRAFFIC_LIGHT_Updata();
         }
}

void TRAFFIC_Lights_Init(const eLight_State START_STATE)
{
  Light_State_G=START_STATE;
}

void TRAFFIC_LIGHT_Updata(void)
{
 switch(Light_State_G)
 {
```

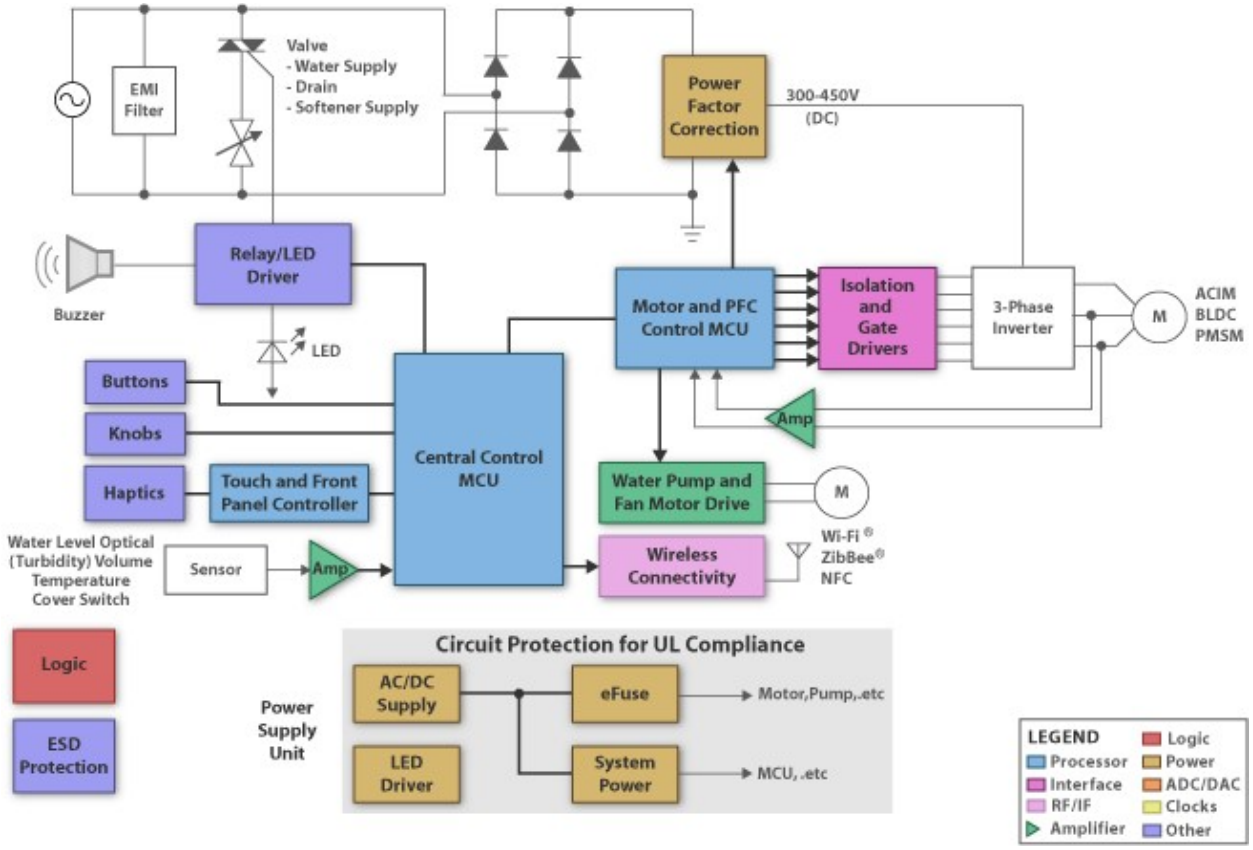
```
   case RED:
         Red_Light=ON;
         Amber_Light=OFF;
         Green_Light=OFF;
         if(++Timer_In_State==RED_DURATION)
         {
          Light_State_G=AMBER;
           Timer_In_State=0;
         }
         break;
          case AMBER:
         Red_Light=OFF;
         Amber_Light=ON;
         Green_Light=OFF;
         if(++Timer_In_State==AMBER_DURATION)
         {
          Light_State_G=GREEN;
           Timer_In_State=0;
         }
         break;
         case GREEN:
         Red_Light=OFF;
         Amber_Light=OFF;
         Green_Light=ON;
         if(++Timer_In_State==GREEN_DURATION)
         {
          Light_State_G=RED;
           Timer_In_State=0;
         }
  }          //swith
} //Update

int main(void)
{
                  Red_Light _TRIS=0;
                  Green_Light _TRIS=0;
                   Amber _Light _TRIS=0;
               TRAFFIC_Lights_Init(RED);
                  TickInit ();
```

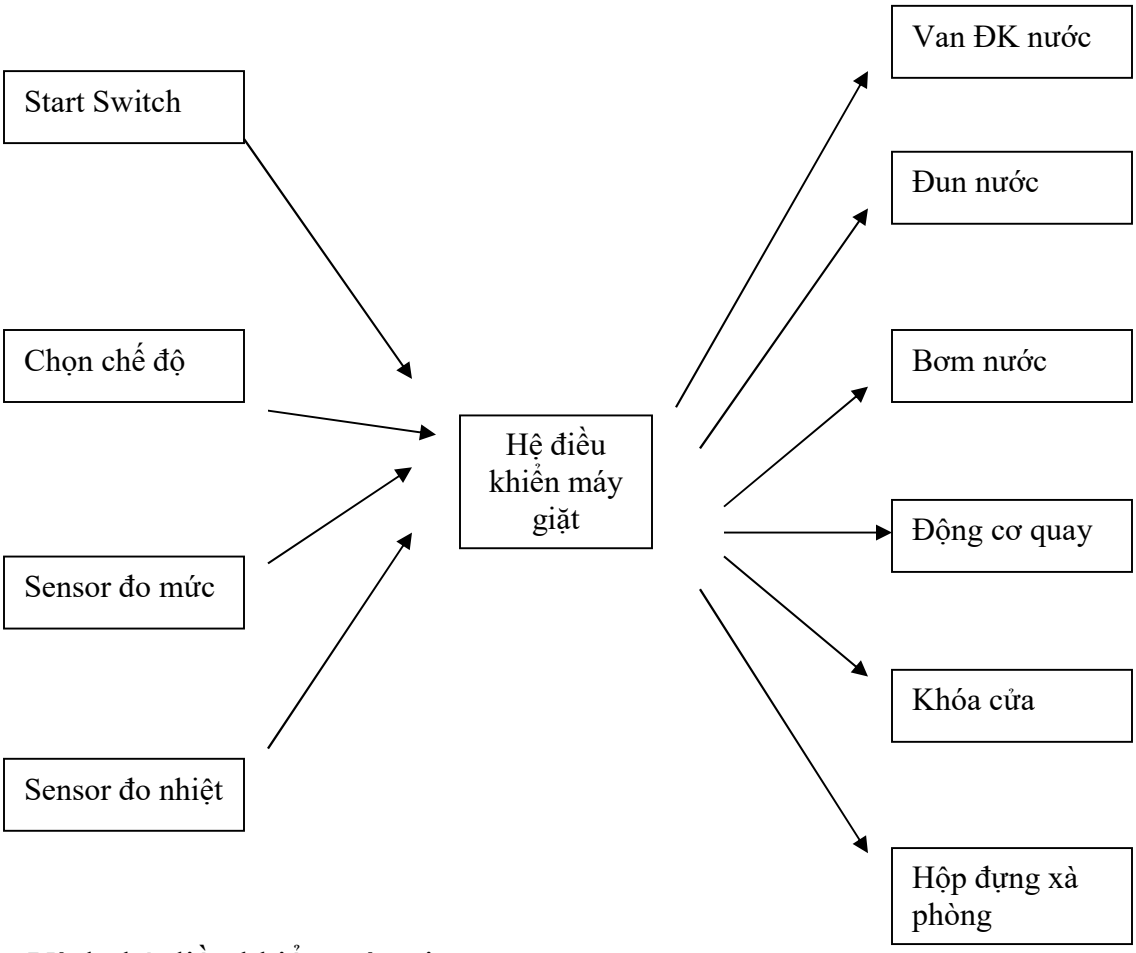
```
        while(1)
        {

        }
}
```

**2. Điều khiển máy giặt.**




Với sự kiện mà trạng thái phụ thuộc vào thời gian và trạng thái đầu vào thì cần thêm điều kiện của trạng thái đầu vào.




Ví dụ hệ điều khiển máy giặt

Các thao tác sử dụng như sau:

- Người sử dụng lựa chọn chế độ giặt:

+ Chế độ 1: giặt có xà phòng, đun nước, thời gian giặt T=60 phút, tốc độ cao.

+ Chế độ 2: giặt có xà phong, đun nước, thời gian giặt T=50 phút, tốc độ thấp.

+ Chế độ 3: giặt không có xà phòng, không đun nước, thời gian giặt T=10 phút, tốc độ thấp.

- Khởi động máy giặt bằng cách ấn nút Start
- Chốt cửa.
- Van nước mở để cho phép nước vào thùng giặt.
- Nếu chế độ có xà phòng thì hộp đựng xà phòng mở.

- Nếu nước đầy thì đóng van cấp nước.
- Nếu giặt ở chế độ nước ấm thì thiết bị đun nước làm việc, khi đạt nhiệt độ thì cắt thiết bị đun nước.
- Động cơ giặt được cấp điện để quay lồng giặt, động cơ quay theo chu trình chiều thuân và chiều nghịch tuần tự 10 giây, với tốc độ khác nhau. Khi đủ thời gian giặt động cơ dừng và chuyển sang chế độ xả nước.
- Ơ chế độ xả nước bơm hút nước hoạt động, khi hút hết nước động cơ bơm cắt, máy giặt về trạng thái ban đầu.

Để xây dựng phần mềm cho hệ điều khiển trước tiên ta định nghĩa các cổng vào ra như sau:

- Các cổng vào:

Công tắc chọn chế độ:

- Chế độ 1: cổng I4
- Chế độ 2: cổng I5
- Chế độ 3: cổng I6

Cổng tắc Start cổng I0

Đo mức nước:

- Mức thấp cổng I1
- Mức cao cổng I2

Đo nhiệt: cổng I3

- Các cổng ra:

Van nước cấp: cổng Q0

Điều khiển hộp xà phòng Q1

Điều khiển nhiệt độ Q2

Điều khiển khóa của Q3

Điều khiển động cơ chạy dừng Q4

Điều khiển chiều quay động cơ Q5

Điều khiển tốc độ nhanh chậm Q6

Điều khiển bơm xả nước Q7

Chương trình điều khiển như sau:

```
#include <p24Fxxxx.h>

#define TIMER_PERIOD            20000
#define TRUE  1
#define FALSE 0

    #define Start                   LATDbits.LATD6
    #define Start_TRIS              TRISDbits.TRISD6

    #define Muc_thap                    LATDbits.LATD5
    #define Muc_thap_TRIS     TRISDbits.TRISD5
    #define Muc_cao             LATDbits.LATD4
    #define Muc_cao _TRIS   TRISDbits.TRISD4

    #define Donhiet                 LATDbits.LATD3
    #define Donhiet _TRIS       TRISDbits.TRISD3

    #define Chedo1                  LATDbits.LATD2
    #define Chedo1_TRIS       TRISDbits.TRISD2
    #define Chedo2              LATDbits.LATD1
    #define Chedo2 _TRIS    TRISDbits.TRISD1
   #define Chedo3                   LATDbits.LATD0
    #define Chedo3_TRIS       TRISDbits.TRISD0
    #define Cap_nuoc                LATCbits.LATC7
    #define Cap_nuoc _TRIS          TRISCbits.TRISC7

   #define Xa_phong                 LATCbits.LATC6
    #define Xa_phong_TRIS    TRISCbits.TRISC6
    #define Nhiet_do            LATCbits.LATC5
    #define Nhiet_do _TRIS  TRISCbits.TRISC4
   #define Khoa                     LATCbits.LATC3
    #define Khoa_TRIS          TRISCbits.TRISC3
    #define DC_Chay                 LATCbits.LATC2
    #define DC_Chay _TRIS          TRISCbits.TRISC2
  #define DC_chieu                  LATCbits.LATC1
    #define DC_chieu _TRIS        TRISCbits.TRISC1
     #define DC_tocdo                 LATCbits.LATC0
    #define DC_tocdo _TRIS          TRISCbits.TRISC0
  #define Xa_nuoc                   LATAbits.LATA7
```

```
    #define Xa_nuoc _TRIS          TRISAbits.TRISA7

bit Co_xaphong,Tocdo_cao,Chieu_thuan,Co_nhiet;
unsigned char Thoigian_giat;

typedef enum{IDE,CAP_NUOC,DUN_NUOC,DONGCO,XANUOC}  eControl_State;

eControl_State Mode;
int count_i,count;
int count_s=0,count_m;

void Init_Port(void)
{
Start_TRIS=1;
Muc_thap_TRIS=1;
Muc_cao_TRIS=1;
Donhiet _TRIS=1;
Chedo1_TRIS=1;
Chedo2_TRIS=1;
Chedo3_TRIS=1;
Xa_phong_TRIS          =0;
Nhiet_do _TRIS=0;
DC_Chay _TRIS=0;
DC_chieu _TRIS       =0;
DC_tocdo _TRIS=0;
Xa_nuoc _TRIS=0;
}

void TickInit( void )  //1uS
{
   TMR4 = 0;
   PR4 = TIMER_PERIOD;// 20000
   T4CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
        GATED_TIME_DISABLED    |    TIMER_16BIT_MODE    |    TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS1bits.T4IF = 0;          //Clear flag
   IEC1bits.T4IE = 0;          //Enable interrupt
   T4CONbits.TON = 0;          //Run timer
```

```
//// Timer 2  10ms
    TMR2 = 0;
   PR2 = 10000;
   T2CON = TIMER_ON | STOP_TIMER_IN_IDLE_MODE | TIMER_SOURCE_INTERNAL |
         GATED_TIME_DISABLED     |     TIMER_16BIT_MODE     |     TIMER_PRESCALER;
//TIMER_PRESCALER8

   IFS0bits.T2IF = 0;            //Clear flag
   IEC0bits.T2IE = 1;            //Enable interrupt
   T2CONbits.TON = 1;            //Run timer
}

void __attribute__((interrupt, shadow, auto_psv)) _T2Interrupt(void)  //10ms
{

   // Clear flag
   IFS0bits.T2IF = 0;
   T2CONbits.TON = 0;
         count++;
         if(count==100)  //1s
         {
          count=0;
     count_s++;
         }
         count_i++;
         if(count_i==6000)  //60s
         {
          count_i=0;
     count_m++;
         }
}

void Maygiat_Init(const eControl_State START_STATE)
{
  Mode=START_STATE;
}
```

```
void  Kiemtra_Chedo(void)
{
 if(Chedo1)
 {
   Co_xaphong=TRUE;
   Co_nhiet=TRUE;
        Tocdo_cao=TRUE;
        Thoigian_giat=60;
 }
 if(Chedo2)
 {
   Co_xaphong=TRUE;
   Co_nhiet=TRUE;
        Tocdo_cao=FALSE;
        Thoigian_giat=50;
 }
 if(Chedo3)
 {
   Co_xaphong=FALSE;
   Co_nhiet=FALSE;
        Tocdo_cao=FALSE;
        Thoigian_giat=10;
 }
}

void MAYGIAT_Updata(void)
{
 switch(Mode)
 {
  case IDE:
        Cap_nuoc=0;
        Xa_phong=0;
        Nhiet_do=0;
        Khoa =0;
        DC_Chay=0;
   DC_chieu=0;
   DC_tocdo=0;
   Xa_nuoc=0;
   Kiemtra_Chedo();
```

```
      if(Start)  //Dieu khien chuyen che do
        Mode=CAP_NUOC;

      break;
       case CAP_NUOC:
      Cap_nuoc=1;
       Khoa=1;
      if(Co_xaphong)
      Xa_phomh=1;
       if(Muc_cao)
        {
          Cap_nuoc=0;
                Mode=DUN_NUOC;
        }
      break;
      case DUN_NUOC:
Nhiet_do=1;
       if(Donhiet)
       {
         Nhiet_do=0;
         Mode=DONGCO;
 count_s=0;
        count_m=0;
       }
      break;
      case DONGCO:
DC_Chay=1;
       if(Donhiet)
        Nhiet_do=0;
       else
       Nhiet_do=1;

       if(Chieu_thuan)
  DC_chieu=0;
        else
         DC_chieu=1;
      if(Tocdo_cao)
       DC_tocdo=1;
```

```
        else
         DC_tocdo=0;
        if(count_s==10)
        {
          count_s=0;
          Chieu_thuan=~Chieu_thuan;
        }

   if(count_m== Thoigian_giat)
        {
          count_m=0;
          DC_Chay=0;
          DC_chieu=0;
          DC_tocdo=0;
          Mode=XANUOC;
        }
        break;
        case XANUOC:
    Xa_nuoc=1;
         if(Muc_thap)
         {
          Xa_nuoc=0;
          Mode=IDE;
         }
        break;
  }          //swith
}  //Update

int main(void)
{
                                Init_Port();
                Maygiat_Init(IDE);
                  TickInit ();
        while(1)
        {
          MAYGIAT_Updata();
        }
}
```

3. Truyền thông UART.

```
#include <p24Fxxxx.h>
   _CONFIG2(IESO_OFF & FNOSC_PRI & FCKSM_CSDCMD & OSCIOFNC_OFF & POSCMOD_HS)
 // _CONFIG1(JTAGEN_OFF & GCP_OFF & GWRP_OFF & COE_OFF & ICS_PGx2 & FWDTEN_OFF)
   _CONFIG1(JTAGEN_OFF & GCP_ON & GWRP_OFF & COE_OFF & ICS_PGx2 & FWDTEN_OFF)

void Init_Port(void)
{
   __builtin_write_OSCCONL(OSCCON & 0xBF);
  // Configure Input Functions (Table 9-1))
  // Assign U2RX To Pin RP8
 //   RPINR18bits.U1RXR = 8;
  RPINR19bits.U2RXR = 8;  //PC
  // Assign U2TX To Pin RP9
 //  RPOR4bits.RP9R = 3;
   RPOR4bits.RP9R = 5;

 //Assign U1RX To Pin RP17
 //  RPINR19bits.U2RXR = 17;  //SIM
  RPINR18bits.U1RXR = 17;  //SIM
// Assign U1TX To Pin RP10
//   RPOR5bits.RP10R = 5;
  RPOR5bits.RP10R = 3;

 // Assign INT1  To Pin RP28
  RPINR0bits.INT1R=28;  //IQR

  //Assign SPI
  RPINR20bits.SDI1R=27;     //MISO->RP27

  RPOR9bits.RP19R = 7;  //MOSI->RP19
  RPOR10bits.RP21R = 8;  //CLK->RP21
  // Lock Registers
 __builtin_write_OSCCONL(OSCCON | 0x40);
```

```
 AD1PCFGL=0xffff;   //All Analog port are Disable
 AD1PCFGH=0xffff;
}

void UARTInit(void)
{

   IFS0bits.U1RXIF=0;
   IEC0bits.U1RXIE=1;//Enable Interrupt U1UART
   IEC1bits.U2RXIE=1;//Enable Interrupt U2UART

   U1MODEbits.UARTEN=1;
   U1STAbits.UTXEN=1;
   U2MODEbits.UARTEN=1;

   U2STAbits.UTXEN=1;

   U1MODEbits.BRGH=1;
   U1BRG=16;         //BaudRate=115200 SIM 968
   U2MODEbits.BRGH=1;    //BaudRate=115200
   U2BRG=16;  //PC

   INTCON2bits.INT1EP=0; //Interrupt on rising edge

}

void __attribute__((interrupt, shadow, auto_psv)) _U2RXInterrupt(void)  //PC
{
   // Clear flag
   char ch;

   ch=U2RXREG&0x00ff;
   IFS1bits.U2RXIF= 0 ;
         if(U2STAbits.OERR)
         {
                  U2STAbits.OERR=0;
                  return;
   }
 if(ch=='R')
```

```
  Mode_PC1=TRUE;

 if(Mode_PC1)
 {
   while(U1STAbits.UTXBF);  /* wait if the buffer is full */
     U1TXREG = ch;
 }
 if(ch=='#')      //Da nhan xong
  Mode_PC1=FALSE;
}

BOOL WriteUART1(char *buffer, unsigned int Len)  //SIM
{
      unsigned int k;
      for(k=0;k<Len;k++)
      {
      //    Timer_Out_UART=0;
          while(U1STAbits.UTXBF)/* wait if the buffer is full */
           {
        //     if (Timer_Out_UART>5)
          //    return TRUE;
           }
          U1TXREG = *buffer++;   /* transfer data byte to TX reg */
      }
 return FALSE;
}

void Send_State_Ok(void)
{
    WriteUART1("STATE,OK",8);
}

int main (void)
{
 Init_Port();
 UARTInit();
send_cstring("Test String");
 while(1)      //Main loop
   {
```

```
    WriteUART1("STATE,OK",8);
}
```

## IV. CẢI THIỆN ĐỘ TIN CẬY CỦA HỆ THỐNG NHỜ WATCHDOG TIMER.

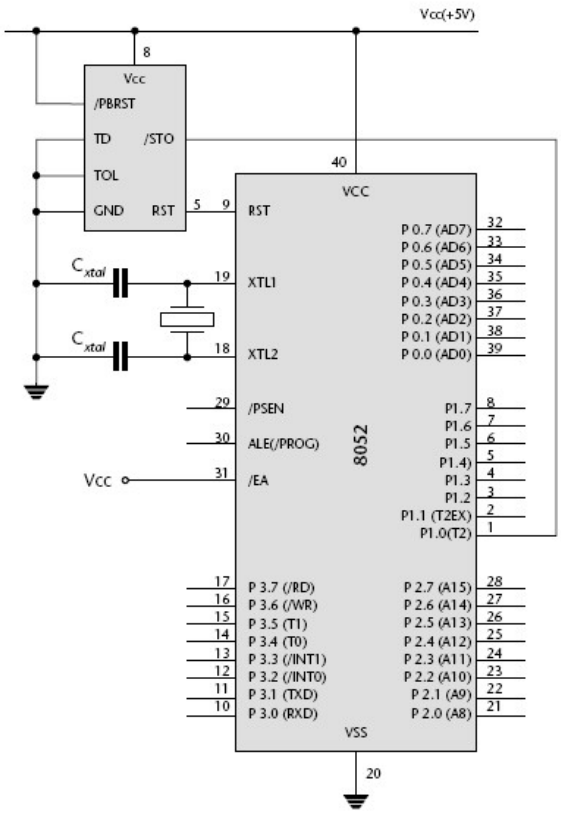
Watch Dog Timer được sử dụng thông thường có 2 đặc tính sau:

- Timer cần phải đươc thường xuyên cập nhật (xóa giá trị đếm) tương ứng với một khoảng thời gian đặt trước.
- Khi khởi động vi xử lý vừa kiểm tra nguyên nhân Reset. Tức là phải kiểm soát được sự khởi động bình thường hay do nguyên nhân Watch Dog Timer bị tràn. Điều này cần thiết  để kiểm soát được lỗi do phần cứng hoặc lập trình. Chi một vài chip xử lý cho phép xác định hệ thống khởi động bình thường hay cho Watch Dog.

Tính ổn định và an toàn:

Trước khi sử dụng 1 Watch Dog, vừa phải khẳng định được dạng Timer này sẽ làm tăng (hơn là giảm) độ tin cậy của ứng dụng.

- Trước tiên xét đến khả năng khôi phục sự cố hệ thống. Cần lưu ý là sự cố này không thường xuyên xảy ra, việc khắc phục sự cố được thực hiện trên nguyên lý sau: Khi hệ thống hoạt động bình thường , tương ứng với trường hợp xảy ra lỗi thì hệ thống sẽ khởi động lại.
- Với 1 thiết kế phần mềm không được kiểm soát hỏng, 1 WatchDog có thể làm giảm tính trên cùng hệ thống như trong 1 số trường hợp sẽ tạo ra 1 tín hiệu Reset hệ thống có tính chu kỳ sau 1 khoảng thời gian nhất định.
- Cũng cần phải lưu ý rằng WatchDog không phù hợp với 1 vài ứng dụng do thời gian xử lý lỗi quá dài. Ví dụ, hệ thống phanh trên ôtô sẽ được khởi động lại nếu có lỗi trong khoảng 500ms, tuy nhiên với khoảng thời gian này xe ôtô có thể đã đi được đến 15m và có thể đã xảy ra tai nạn. Như vậy trong trường hợp này phải sử dụng những giải pháp khác để nâng cao tính tin cậy của hệ thống.




Để thực hiện tính năng WatchDog cần xây dựng chương trình theo các bước sau:

- Xây dựng 1 hàm ngắt Timer.
- Xây dựng 1 hàm làm mới dữ liệu.

Ví dụ, ta khởi tạo 1 Timer có thời gian trên là 60ms. Trong trường hợp bình thường, Timer này không bao giờ bị tràn, cho hàm làm mới luôn được gọi trong khoảng thời gian này và nhiệm vụ của nó là khởi tạo lại Timer. Nếu vì 1 lý do nào đó, chương trình bị lỗi và hàm làm mới không được gọi, khi Timer bị tràn thì 1 hàm ngắt sẽ được gọi thực hiện vệc xử lý lỗi.

Phương pháp sử dụng có thể là WatchDog mềm và cứng:

- Với WatchDog mềm ta xây dựng 1 hàm xử lý
- Với WatchDog cứng thì hệ thống sẽ tự khởi tạo bởi 1 khối phần cứng.

```
/*-------------------------------------------------------------------*-

   Main.C (v1.00)

------------------------------------------------------------------

   Framework for a central heating system using 'Super Loop'.

   *** Assumes external '1232' watchdog timer on P1^0 ***

-*-------------------------------------------------------------------*/

#include "Cen_Heat.h"
#include "Dog_1232.h"

/*-------------------------------------------------------------------*/
void main(void)
   {
   // Init the system
   C_HEAT_Init();

   // Watchdog automatically starts

   while(1)
      {
      // Find out what temperature the user requires
      // (via the user interface)
      C_HEAT_Get_Required_Temperature();

      // Find out what the current room temperature is
      // (via temperature sensor)
      C_HEAT_Get_Actual_Temperature();

      // Adjust the gas burner, as required
      C_HEAT_Control_Boiler();

      // Feed the watchdog
      WATCHDOG_Feed();
      }
   }

/*-------------------------------------------------------------------*-
  -------- END OF FILE ------------------------------------------
-*-------------------------------------------------------------------*/
```

```
/*---------------------------------------------------------------------*-

   Dog_1232.C (v1.00)

  ---------------------------------------------------------------------

   Watchdog timer library for external 1232 WD.

-*---------------------------------------------------------------------*/

#include "Dog_1232.h"

#include "Main.h"

// ------ Port pins -----------------------------------------------

// Connect 1232 (pin /ST) to the WATCHDOG_pin
sbit WATCHDOG_pin = P1^0;

/*---------------------------------------------------------------------*-

   WATCHDOG_Feed()

   'Feed' the external 1232-type watchdog chip.

-*---------------------------------------------------------------------*/
void WATCHDOG_Feed(void)
   {
   static bit WATCHDOG_state;

   // Change the state of the watchdog pin
   if (WATCHDOG_state == 1)
      {
      WATCHDOG_state = 0:
      WATCHDOG_pin = 0;
      }
   else
      {
      WATCHDOG_state = 1;
      WATCHDOG_pin = 1;
      }
   }

/*---------------------------------------------------------------------*-
  ------- END OF FILE -------------------------------------------
-*---------------------------------------------------------------------*/
```

# CHƯƠNG 4. XÂY DỰNG CHƯƠNG TRÌNH ĐA NHIỆM

## 4.1 Cấu trúc chương trình đa nhiệm

Để thực hiện 1 bài toán trong chương trình xử lý nhiều lúc cần thực hiện nhiều nhiệm vụ. Các nhiệm vụ được phân bố vào một danh sách thực hiện (Scheduler). Danh sách thực hiện được chia thành các dạng sau:

1. Cấu trúc đơn nhiệm, cấu trúc này có đặc điểm sau:

- Được thực hiện tại những thời điểm định trước (có chu kỳ hoặc không có chu kỳ)
- Khi 1 nhiệm vụ (thẻ sự kiện) hoạt động, thẻ sự kiện khác sẽ đưa vào danh sách đợi.

- Khi CPU ở trạng thái rỗi, thẻ sự kiện kế tiếp trong danh sách đợi sẽ thực hiện
- Khi 1 thẻ sự kiện hoạt động xong sẽ trao lại quyền điều kiển cho danh sách thực hiện.

Để xây dựng 1 danh sách thực hiện cần chú ý như sau:

- Danh sách thực hiện và thẻ sự kiện phái ngắn gọn chiếm ít câu lệnh
- Thủ tục cấp phát bộ nhớ cho 1 thẻ sự kiện thực hiện tại 1 thời điểm.
- Thông thường chương trình được xây dựng dựa trên ngôn ngữ bậc cao.
- Xây dựng cấu trúc của danh sách thực hiện.
- Hàm khởi tạo
- Hàm ngắt để thay đổi danh sách tương ứng với quãng thời gian định trước.
- Hàm bổ xung thẻ sự kiện vào danh sách.
- Hàm cấm thực hiện thẻ sự kiện khi nó không tích cực.
- Hàm xóa sự kiện ra khỏi danh sách.

2. Cấu trúc đa nhiệm:

Chương trình có cấu trúc như sau:

- Các thẻ sự kiện trong danh sách sẽ hoạt động trong thời gian nhất định (có chu kỳ hoặc không có chu kỳ).
- Khi đang có 1 thẻ sự kiện hoạt động thì nó sẽ được đẩy vào danh sách đợi
- Thẻ sự kiện sẽ hoạt động với một thời gian cố định, nếu chưa kết thúc nó sẽ được tạm dừng và đẩy ngược trở lại danh sách đợi. Thẻ sự kiện kế tiếp đang đợi sẽ được hoạt động trong khoảng thời gian cố định và quá trình tiếp diễn.

Các đặc điểm cần lưu ý khi thực hiện cấu trúc này:

- Thực hiện cấu trúc đa nhiệm tương đối phức tạp do cần thực hiện 1 thông báo (Semaphores) để loại trừ xung đội các sự kiện đồng thời cùng sử dụng nguồn tài nguyên chia sẻ.
- Cần cấp phát bộ nhớ để giữ trạng thái mà sự kiện trước đó đang thực hiện.
- Chương trình cần xây dựng ít nhất 1 phần bằng ngôn ngữ Assembler.
- Chương trình cần được tạo ra bởi 1 ứng dụng chuyên dụng.

Về mặt tốc độ dạng này có thể đáp ứng được với các sự kiện bên ngoài nhanh

Về tính tin cậy và độ ổn định kém hơn sơ với chương trình đơn nhiệm.

Với chương trình theo cấu trúc đa nhiệm phải đặc biệt lưu ý các trường hợp sau dễ dẫn đến xung đột:
- Đoạn lệnh thay đổi hoặc đọc giá trị các biến, đặc biệt là các biến tổng thể dùng cho truyền thông.
- Đoạn lệnh giao tiếp với phần cứng, dạng như các cổng, bộ biến đổi ADC...
- Đoạn lệnh gọi 1 hàm chung.

Với 1 chương trình đơn nhiệm các vấn đề này không xảy ra do chỉ 1 thẻ sự kiện được tích cực tại 1 thời điểm.

Để loại trừ các trường hợp xấy kể trên với chương trình đa nhiệm ta thực hiện 2 giải pháp sau:
- Tạm dừng việc lập danh sách bằng cách cấm thực hiện ngắt danh sách trước khi bắt đầu thực hiện đoạn lệnh tranh chấp, cho phép trở lại ngắt khi đã ra khỏi đoạn lênh tranh chấp.
- Hoặc sử dụng 1 khóa (hoặc 1 dạng khác của việc truyền thông báo)

để thực hiện chức năng tương tự như trên.

Giải pháp đầu tiên được thực hiện như sau, khi bắt đầu truy nhập vào tài nghuyên chia sẻ ta cấm việc thực hiện bảng danh sách.

3. Cấu trúc chương trình lai: là một dạng hạn chế của chương trình đa nhiệm.

Có các đặc điểm sau:
- Hỗ trợ số lượng bất kỳ số thẻ sự kiện đơn nhiệm
- Hỗ trợ chỉ 1 thẻ tiến trình đa nhiệm (nó có thể ngắt các tiến trình đơn nhiệm)

Cần lưu ý 1 số vấn đề sau khi thực hiện cấu trúc lai:
- Danh sách thực hiện là dạng đơn giản và có thể thực hiện với đoạn mã ngắn gọn
- Danh sách thực hiện cần cấp phát bộ nhớ cho 2 thẻ sự kiện tại 1 thời điểm
- Chương trình sẽ được tạo ta bằng ngôn ngữ bậc cao.

- Có thể dùng đoạn mã phát triển để tạo ra chương trình.

Về tính ổn định và tin cậy: với một thiết kế cẩn thận thì nó có thể tin cậy như chương trình đơn nhiệm.

## 4.2. Tổng quan chung về hệ điều hành thời gian thực.

### 1. Mô tả và cách thức thực hiện 1 chương trình

Trong phần này mô tả cấu trúc của 1 chương trình, các bước chuẩn bị cho 1 chương trình chạy và chương trình chạy thực thế như thế nào.

**Biên dịch và liên kết**

Trước tiên ví dụ một chương trình đơn giản hiển thị từ Hello World

```
#include <stdio.h>

const char * Text = "Hello World\n";

char Data[] = "Hello Data\n";

int Uninitialized;   // Bad Practice

int main(int argc, char * argv[])
{
     printf(Text);
}
```

Chương trình trên ngôn ngữ C++ in từ Hello World sau đó xuống dòng trên màn hình. Trước khi thực hiện cần chuyển chương trình về dạng mã máy thực hiện trên máy tính. Quá trình chuyển đổi này thực hiện theo 2 bước: biên dịch (Compilation) và liên kết (Linking).

Trong bước thứ nhất, biên dịch được thực hiện bởi chương trình có tên Complier. Chương trình này sẽ sử dụng file chương trình nguồn dạng text ví dụ Hello.cc biên dịch sang 1 file khác ví dụ là Hello.o. Lênh thực hiện chương trình này như sau:

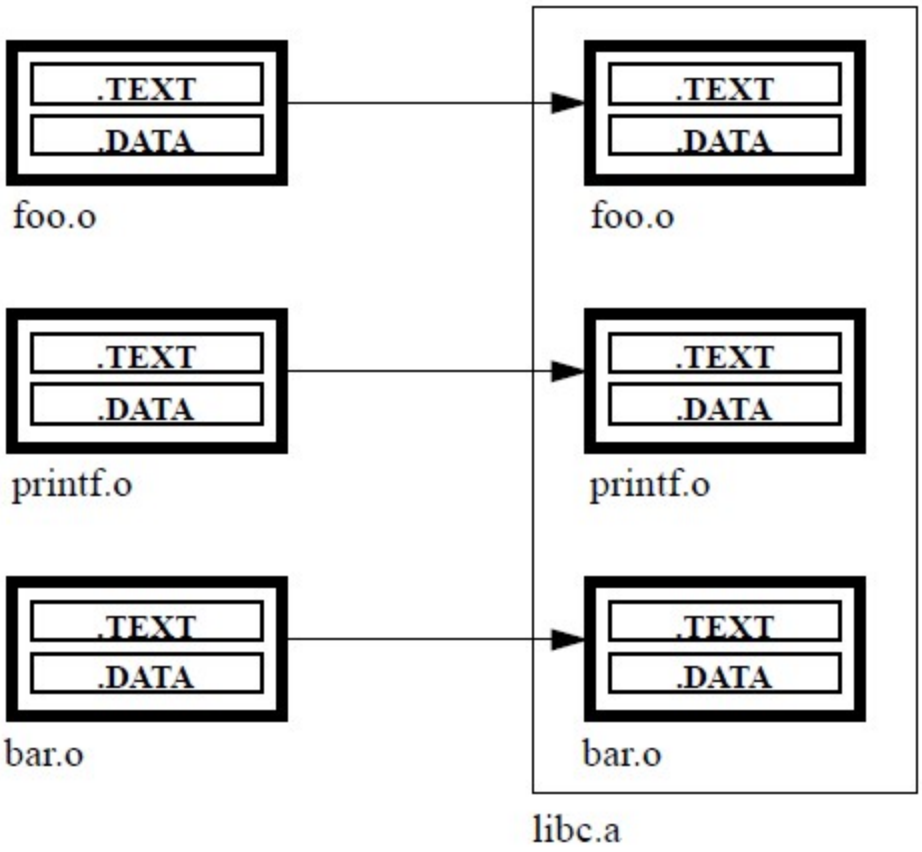
g++ -o Hello.o Hello.cc

Trong đó g++ là tên chương trình dịch, Hello.o là file đối tượng (object file) cấu trúc bên trong gồm có 2 phần : TEXT, DATA và BSS. Trong chương trình có dòng lệnh #include <stdio.h> có nghĩa là copy nội dung file stdio.h vào đoạn lệnh

này trước khi biên dịch (preprocessing). Nếu không đưa dòng lệnh này vào thì tại hàm printf sẽ báo sai cú pháp do không tìm thấy khai báo hàm. Hình dưới thể hiện cấu trúc của file Hello.o


Hình 1.1 Cấu trúc file Hello.o

Một vài file đối tượng được tập hợp vào trong 1 file duy nhất đuwọc gọi là thư viện. Một thư viên quan trong là libc.a (tên thư viên có thể khác nhau với hệ điều hành sử dụng): trong thư viên này chưa đoạn mã cho hàm printf trong ví dụ trên đồng thời chưa nhiều hàm chức năng khác. Hình dưới mô tả file libc.a đuwọc tập hợp từ các file đối tượng khác.



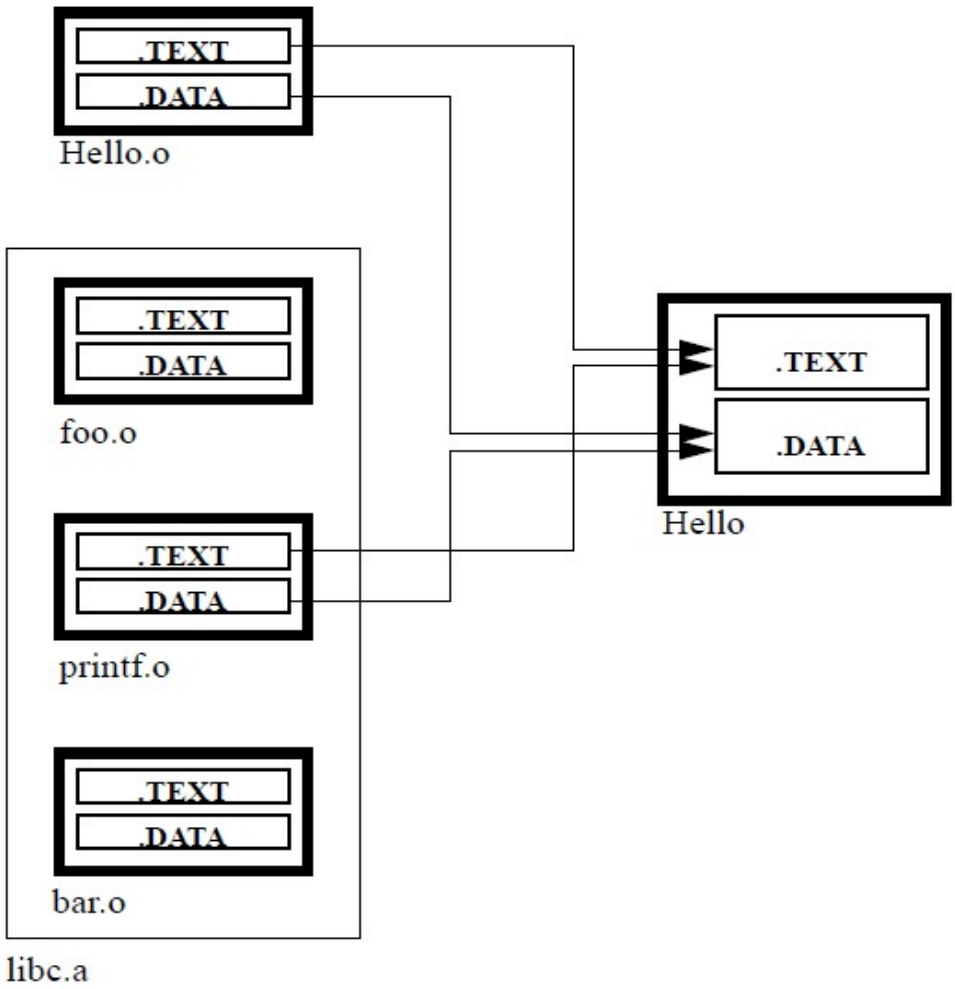

Hình 1.2

Bước thứ 2 của việc chuyển đổi chương trình từ file nguồn thành chương trình chạy là liên kết (link). Ví dụ sử dụng dòng lệnh sau:

Ld –o Hello Hello.o

Quá trình liên kết đuwọc thể hiện ở hình dưới. Trong ví dụ hàm printf được sử dụng trong hàm main  nhưng được khai báo trong file printf.o và tịch hợp vào thư viên libc.a

Quá trình  liên kết sẽ liên kết các file đối tượng  bao gồm 2 đoạn (Section) TEXT và DATA thành 1 file duy nhất cũng có 2 thành phần TEXT và DATA. Quá trình liên kết cũng có thể liên kết cả file thư viện trong trường hợp chương trình sử dụng tới nó. Kết quả của quá trình liên kết có thể đưa ra các định dạng các nhau như HEX Intel, Motorola S …




Hình 1.3

## 2. Nạp và thực hiện chương trình

Sau khi chương trình đuwọc dịch và liên kết, nó có thể thực hiện. Phụ thuộc vào môi trường thực hiện là ở vi xử lý hay máy tính PC có cách thực hiện khác nhau như bảng dưới

| TT | Thực hiện tại máy tính PC | Thực hiện tại vi xử lý |
|---|---|---|
| 1 | Đoạn TEXT của chương trình được nạp vào bộ nhớ chương trình ( Bộ nhớ RAM của máy tính) | Đoạn TEXT nằm sẵn ở bộ nhớ chương trình (EPROM, FLASH) của vi xử lý |
| 2 | Phụ thuộc vào định dạng của quá trình liên kết, địa chỉ của đoạn TEXT co thể được phân bố lại. | Địa chỉ của đoạn TEXT được tính toán và phân bố trong quá trình kiên kết |
| 3 | Đoạn DATA của chương trình được nạp vào bộ nhớ chương trình (bộ nhớ RAM của máy tính) | Đoạn DATA của chương trình nằm sẵn ở bộ nhớ chương trình (EPROM, FLASH) của vi xử lý |
| 4 | Phụ thuộc vào định dạng của quá trình liên kết, địa chỉ của đoạn TEXT co thể được phân bố lại. | Đoan DATA trong quá trình thực hiện được lưu trữ trong RAM |

Như vậy việc thực hiện chương trình trong hệ thống nhúng (vi xử lý) dễ dàng hơn trên máy tính.

## 3. Quyền ưu tiên chế độ đa nhiệm

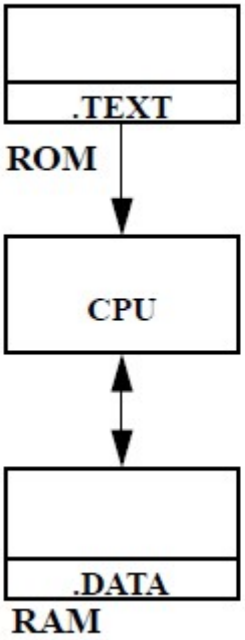
Trong phần trước mô tả việc thực hiện của một chương trình tại một thời điểm. Nhưng trong trường hợp cần thực hiện một vài chương trình cùng lúc (đa nhiệm) thì cần xem xét đến quyền ưu tiên của các chương trình.

Ta định nghĩa, một tác vụ (task) là 1 chương trình thực hiện và đa nhiệm (multitasking) là một vài tác vụ thực hiện song song.

Khái niệm về quyền ưu tiên đa nhiêm sẽ được mô tả ở phần sau

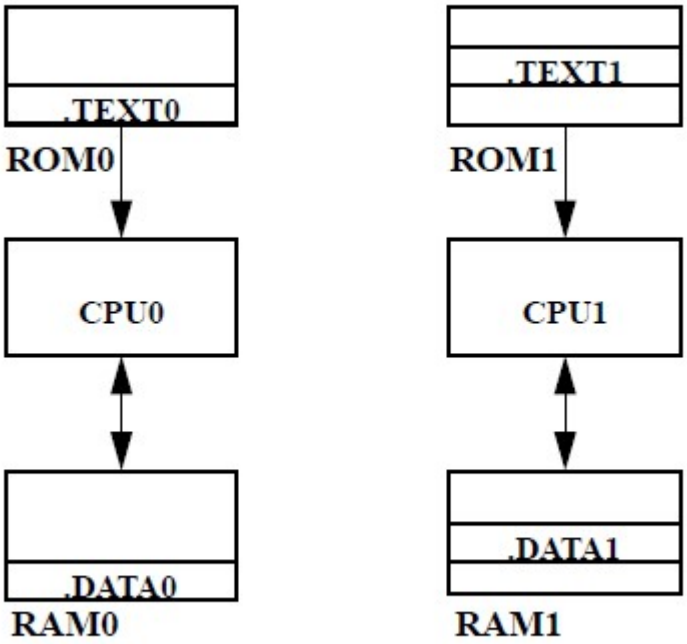
## 4. Phần cứng thực hiện chương trình

Giả sử ta dùng một CPU với bộ nhớ chương trình (ROM,FLASH) và mộ nhớ dữ liệu RAM. CPU có thể đọc và thực hiện lệnh từ bộ nhớ chương trình và đọc,ghi vào bộ nhớ dữ liệu. Các lệnh này chi là đoạn (section) TEXT ở phần trên .




Hình 1.4

Giả sử ta có 2 chương trình cần thực hiện song song. Giải pháp trong trường hợp này là tăng thêm 1 CPU để thực hiện chương trình thứ hai như hình dưới




Hình 1.5

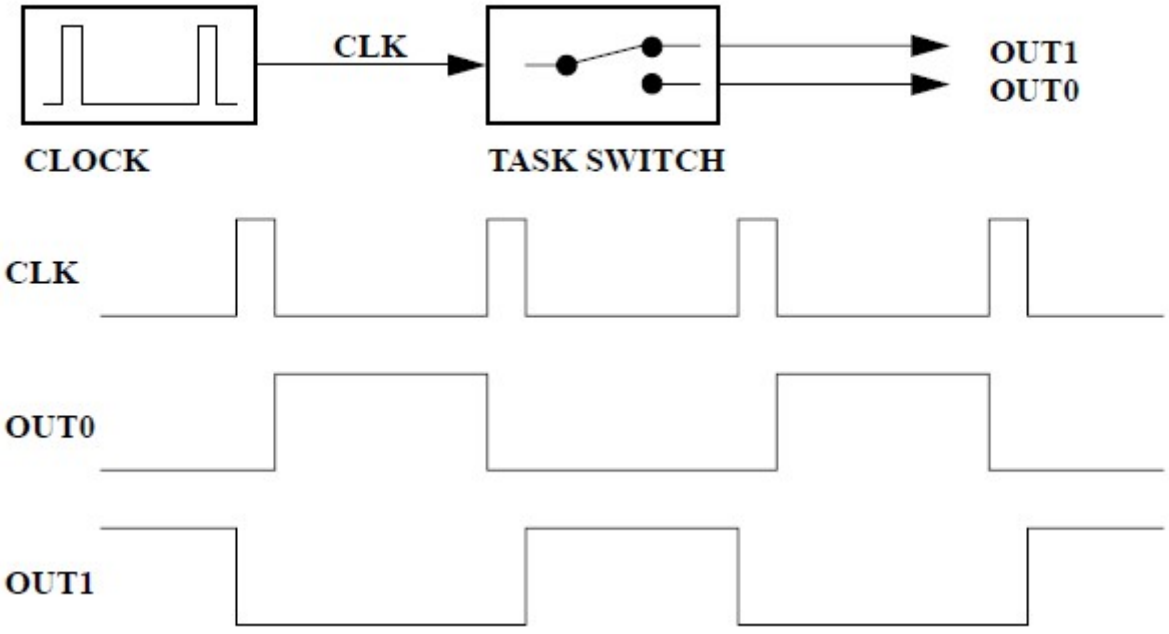
Ta thấy trong trường hợp này đoạn TEXT và DATA được bố trí ở 2 vùng khác nhau ở ROM và RAM.

Bởi vì cần tăng thêm CPU nên chi phí sẽ tăng lên và không phải là giải pháp tối ưu. Bảng dưới phân tích ưu nhược điểm khi tăng số CPU

| Ưu điểm | Nhược điểm |
|---|---|
| Hai chương trình được bảo vệ tránh sự ảnh hưởng lẫn nhau. Nếu một chương trình có lỗi cũng không ảnh hưởng đến cái kia | Cần sử dụng thêm CPU, ROM và RAM |
| | Hai chương trình không thể liên hệ dữ liệu với nhau |

## 5. Chuyển tác vụ

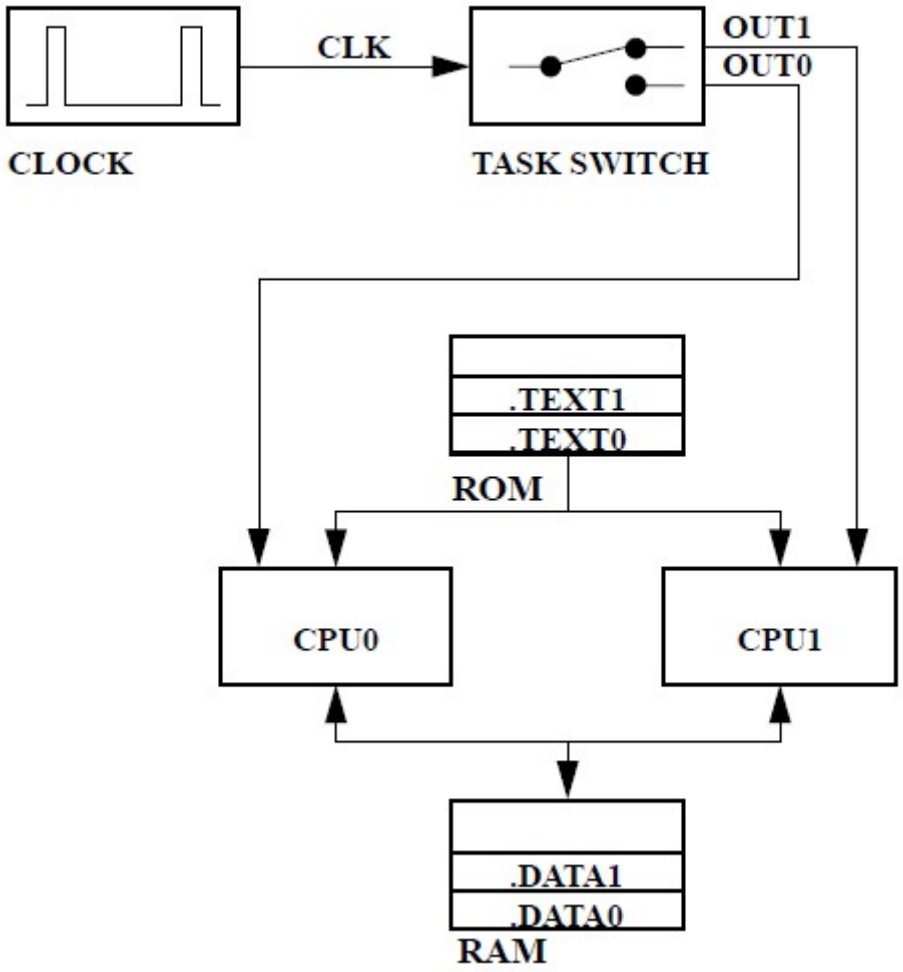
Sử dụng một mô hình khác là bỏ 1 bộ nhớ ROM và một bộ nhớ RAM, vẫn sử dụng 2 CPU để truy nhập vào 1 ROM và 1 RAM. Bổ sung thêm bộ phần cho phép CPU nào hoạt động bằng cách tín hiệu cho phép vào từng CPU theo các sơ đồ sau:




Hình 1.6

Mỗi một CPU có 1 đầu vào cho phép CPU chuyển sang trạng thái chạy hay dừng. Nếu đầu vào tích cực. CPU ở chế độ hoạt động bình thường. Nếu đầu vào

không tích cực, CPU sẽ thực hiện nốt chu kỳ máy hiện tại và ngắt  kết nối với ROM, RAM. Bằng cách này, tại 1 thời điểm chỉ có 1 CPU được kết nối với ROM và RAM, trong khi đó CPU kia sẽ ở trạng thái nghỉ và không kết nối ROM, RAM. Như vậy ta có thể giảm bớt ROM và RAM khi sử dụng.



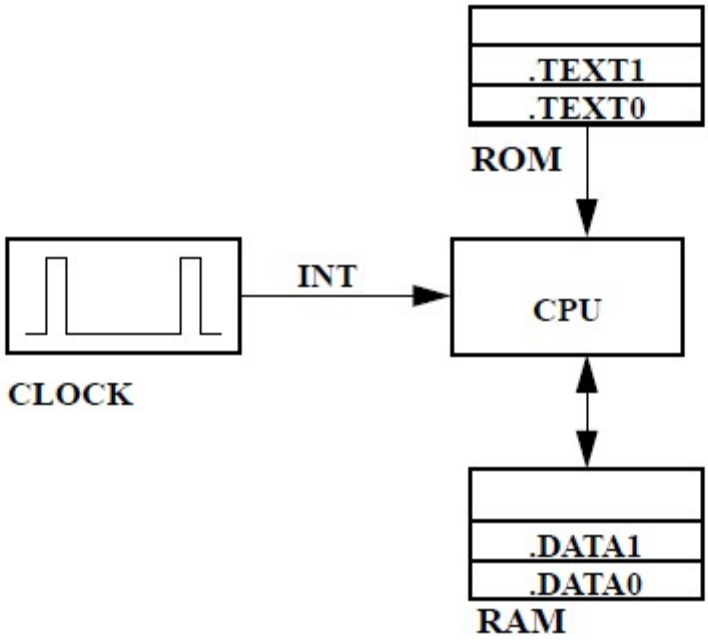

Hình 1.7

Khi sử dụng cùng chung bộ nhớ chia sẻ RAM, hai vi xử lý có thể liên lạc với nhau.  Tuy nhiên các CPU sẽ không còn ở chế độ bảo vệ (không xung đột nữa) do có thể xung đột khi truy nhập vùng nhớ RAM

### 6. Khối kiểm soát tác vụ

Trong mô hình này không sử dụng 2 CPU để xử lý 2 tác vụ nữa mà ghép 2 tác vụ xử lý trên cùng một CPU. Giải pháp này thực hiện không đơn giản do trên một

vi xử lý không thể chia thành 2 đoạn (section riêng biệt). Trước khi giải quyết vấn đề này, ta xem xét mô hình sau:




Hình 1.8

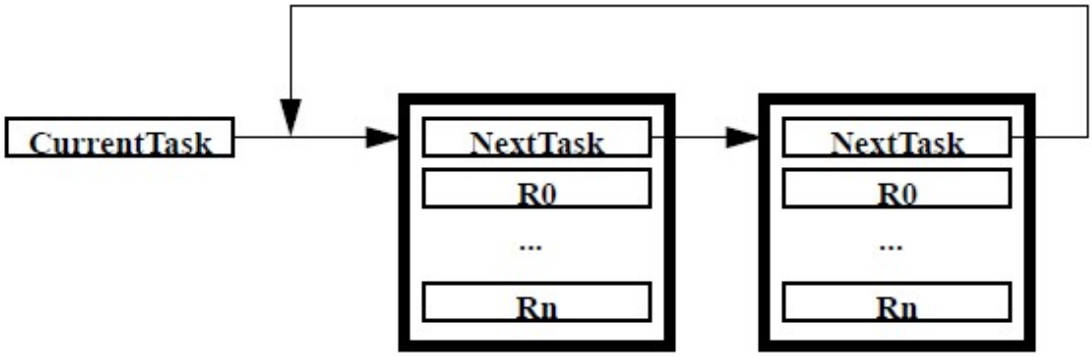
So sánh với sơ đồ trước, ta thấy trong trường hợp này chỉ có 1 CPU và không có khối chuyển tác vụ, thay thế nó là tín hiệu xung nhịp vào chân INT.

Sơ đồ trên gần giống sơ đồ ban đầu hình 1.4 chỉ bổ sung thêm tín hiệu xung nhịp và chân ngắt INT của vi xử lý. Sơ đồ này cho phép thực hiện song song 2 tác vụ.

Với sơ đồ sử dụng 2 CPU và mạch chuyển tác vụ, ta thấy 2 CPU khi làm việc sẽ có những trạng thái khác nhau. Các trạng thái này được lưu trữ bên trong các thanh ghi của CPU và kiểm soát bởi chương trình thực hiện của từng CPU. Để giảm bớt 1 CPU ta cần thay thế mạch chuyển tác vụ cứng bằng thuật toán phần mềm. Khi việc chuyển tác vụ hoạt động, trạng thái của 1 CPU sẽ được cất giữ và trạng thái của CPU thứ hai sẽ được khôi phục hoạt động. Như vậy nó sẽ thực hiện theo thuật toán sau:

- Ghi và lưu trữ thanh ghi bên trong của CPU0
- Phục hồi thanh ghi bên trong của CPU1

Tuy nhiên thuật toán này không áp dụng được cho mô hình ở hình 1.8 do ta chỉ có 1 CPU. Để thay thế mô hình có 2 CPU ở đây ta sử dụng 1 cấu trúc dữ liệu gọi là TCB –Khối kiểm soát tác vụ để mô tả các CPU trong hệ thống. Trong khối TCB có vùng không gian để lưu giữ nội dung các thanh ghi của CPU, ví dụ tư R0 đến Rn. Ngoài ra mỗi một TCB có con trỏ chỉ đến khối TCB mô tả CPU tiếp theo. Nội dung của khối TCB được trình bày ở hình dưới:




Kết quả là thực hiện chuyển tác vụ thông qua hàm dịch vụ ngắt (ISR) như sau:

- Xóa ngắt (nếu cần)
- Lưu trữ thanh ghi bên trong CPU vào khối TCB mà Tác vụ hiện tại (CurrentTask) đang chỉ tới.
- Thay thế Tác vụ hiện tại (CurrentTask) bằng con trỏ Tác vụ kế tiếp (NextTask). Tác vụ kế tiếp này nằm trong khối TCB mà tác vụ hiện tại đang chỉ đến.
- Phục hồi các thanh ghi bên trong CPU từ khối TCB mà Tác vụ hiện tại (CurrentTask) đang chỉ đến.
- Trở về từ ngắt.

Lưu ý rằng dịch vụ ngắt (ISR) không tự thay đổi trạng thái của CPU khi huyển tác vụ. Nhưng dịch vụ ngắt này phục vụ cho chế độ đa nhiệm có ưu tiên (preemplive multitasking). Bằng cách đưa vào khối TCB được định hướng bởi con trỏ Tác vụ kế tiếp (NextTask), mô hình mới có thể cho hoạt động với số lượng Tác vụ bất kỳ.

Có một điểm quan trọng với mô hình này là: Mỗi khi một tác vụ được xem như là Tác vụ hiện tại, nó sẽ luôn tìm đến khối TCB tương ứng với nó. Nếu Tác vụ

hiện tại (CurrentTask) không chỉ đến một Task bất kỳ nào đó thì tác vụ đó không tích cực tại thời gian này.

**7. Xóa lập lịch (De-Scheduling)**

Như ví dụ trên ta có 2 tác vụ hoạt động sẽ chiếm dụng thời gian hoạt động của CPU là bằng nhau. Nếu trong mỗi tác vụ đều tận dụng tối đa năng lực của CPU thì không cần thiết phải thay đổi phân bố thời gian hoạt động của CPU. Với hệ thống nhúng thường xảy ra trường hợp trong khi hoạt động tác vụ cần đợi một sự kiện nào đó mới thực hiện tác động nào đó. Ví dụ, tác vụ kiểm tra trạng thái của phím bấm chức năng, khi phát hiện ra sự ấn phím thì sẽ thực hiện hàm tính toán nào đó (hàm lic).

```
task_0_main()
{
     for  (;;)
          if (button_0_pressed()) lic_0();
}

task_1_main()
{
     for  (;;)
          if (button_1_pressed()) lic_1();
}
```

Việc chuyển tác vụ được điều kiển bới xung nhịp thời gian, mỗi tác vụ chiếm 50% thời gian của CPU, không phụ thuộc vào phím có được ấn hay không.

Như vậy thời gian của CPU sẽ bị lãng phí với tác vụ khi đợi phím bấm và không làm gì cả, tình huống này được gọi là chế độ đợi (busy wait).

Để tối ưu hóa hiệu quả của thời gian CPU, ta sẽ bổ sung một hàm **DeSchedule()** có nhiệm vụ thoát khỏi tác vụ giải phóng thời gian làm việc của CPU khi phím không ấn:

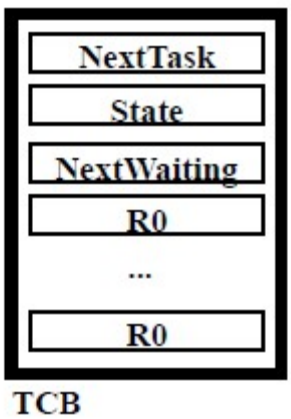
```
task_0_main()
{
     for  (;;)
          if (button_0_pressed()) lic_0();
          else                    DeSchedule();
}

task_1_main()
{
     for  (;;)
          if (button_1_pressed()) lic_1();
          else                    DeSchedule();
}
```

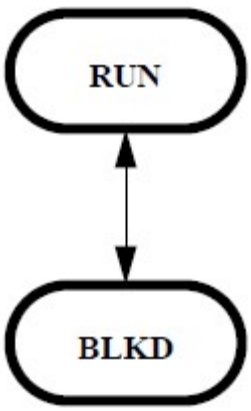
8. Quyền sử dụng tài nguyên (**Semaphores**)

Để tăng khả năng sử dụng hiệu quả thời gian làm việc của CPU và cải thiện thời gian cho việc chuyển tác vụ, người ta đưa thêm một cấu trúc dữ liệu nữa là Semaphores. Các Semaphore này cho phép thay đổi trạng thái của các tác vụ.

Trong mô hình của ta, hai tác vụ sẽ làm việc lần lượt và chiếm một khoảng thời gian của CPU. Với Semaphore sẽ đưa thêm 2 biến vào khối TCB là biến Trạng thái (State) và biến đợi chuyển tiếp (NextWaiting). Thông thường khi khởi tạo State được cài ở chế độ chạy (RUN) và NextWaiting có giá trị 0. Khi cần State có thể được cài giá trị BLKD (blocked – bị khóa), như vậy tác vụ có 2 trạng thái là chạy (RUN) và khóa (BLOCKED) và các biến điều khiển tương ứng với nó.



TCB



Tiếp theo, ta cần thay đổi hàm chuyển tác vụ ISR, trong đó bỏ qua tác vụ khi đang ở trạng thái khóa (BLKD) như sau:

- Xóa ngắt (nếu cần)
- Lưu trữ thanh ghi bên trong CPU vào khối TCB mà Tác vụ hiện tại (CurrentTask) đang chỉ tới.
- Lặp lại:

  Thay thế Tác vụ hiện tại (CurrentTask) bằng con trỏ Tác vụ kế tiếp (NextTask). Tác vụ kế tiếp này nằm trong khối TCB mà tác vụ hiện tại đang chỉ đến.

  Cho đến khi trạng thái của Tác vụ hiện tại là RUN
- Phục hồi các thanh ghi bên trong CPU từ khối TCB mà Tác vụ hiện tại (CurrentTask) đang chỉ đến.
- Trở về từ ngắt.

Có một điểm quan trọng với mô hình này là: Khi xem xét một tác vụ, cần xác định trạng thái của tác vụ, tác vụ hoạt động khi nó ở trạng thái RUN. Trạng thái (State) của tác vụ có thể thay đổi bất kỳ vào thời gian nào, nếu nó không ở trạng thái RUN thì không thể chuyển đến tác vụ này. Một Semaphore sẽ điều khiển trạng thái của Tác vụ.

Khái niệm về Semaphore được hiểu như sau: Một Semaphore mô tả số lượng khả thi tài nguyên hệ thống, Semaphore đếm số lượng tài nguyên. Nếu hiện tại không còn tài nguyên trống, semaphore sẽ đếm số lượng tác vụ đợi đến lượt sử dụng tài nguyên, hay nói cách khác đây chính là số lượng tài nguyên còn thiếu. Nếu nguồn tài nguyên còn thiếu thì khối TCB của tác vụ đang đợi sẽ được bổ sung vào danh sách tác vụ đợi và nó sẽ nằm ở vị trí đầu tiên trong danh sách.

Semaphore bao gồm 2 biến là một bộ đếm và con trỏ tới khối TCB. Trong khối TCB con trỏ đợi kết tiếp (NextWaiting) chỉ hoạt động nếu giá trị bộ đếm nhỏ hơn 0, nếu không thì con trỏ không hoạt động và nó có giá trị 0. Con trỏ mô tả trạng thái của Semaphore thể hiện ở bảng dưới.

| Giá trị bộ đếm | Con trỏ đợi kế tiếp trong khối TCB | Trạng thái |
|---|---|---|
| N>0 | 0 | N nguồn tài nguyên chưa |

| | | |
|---|---|---|
| | | sử dụng |
| N=0 | 0 | Không có nguồn tài nguyên cho phép sử dụng và chưa có tác vụ đợi để sử dụng tài nguyên |
| N<0 | Tác vụ kế tiếp cần đợi giải phóng tài nguyên | N tác vụ đợi tài nguyên, có nghĩa là N tài nguyên còn thiếu |

Khi 1 Semaphore được tạo ra, bộ đếm được gán với giá trị N≥0 tương ứng với N nguồn tài nguyên có thể sử dụng, lúc này con trỏ đợi kết tiếp (NextWaiting) có giá trị 0. Khi đó các tác vụ có thể yêu cầu 1 tài nguyên bằng cách gọi 1 hàm P(), hoặc các tác vụ có thể giải phóng tài nguyên bằng cách gọi 1 hàm V(). Tên của hàm P và V được đề xuất của Dijkctra, người đưa ra khái niệm Semaphore.

Thuật toán của hàm P() như sau:

- If bộ đếm Counter >0 (tức là có dư nguồn tài nguyên )
  (giảm số lượng tài nguyên)
- Else (tức là không còn tài nguyên có dư)
  Giảm bộ đếm (tăng số lượng tác vụ đợi)
  Gán trạng thái (State) của
  Tác vụ hiện tại về trạng thái BLKD
  Bổ sung Tác vụ hiện tại vào phần cuối của
  Danh sách đợi và thực hiện hàm DeSchedule().

Như vậy hàm P kiểm tra giá trị bộ đếm Counter xem có tồn tại tài nguyên dư hay không. Nếu có thì tài nguyên sẽ được sử dụng và sô lượng tài nguyên dư sẽ giảm 1 giá trị trong quá trình thực hiện. Nếu không, số lượng tác vụ đợi sẽ tăng 1 giá trị (đồng nghĩa với giá trị bộ đếm giảm), tác vụ sẽ bị khóa và được bổ sung vào hàng đợi, cuối cùng thực hiện hàm DeSchedule() để việc khóa tác vụ có hiệu lực. Rõ ràng là bộ đếm giám 1 giá trị trong bất kỳ trường hợp nào.

Ta có thể thực hiện thuật toán tương đương như sau:

- Giảm bộ đếm

- Nếu Bộ đếm <0
  Gán trạng thái (State) của
  Tác vụ hiện tại về trạng thái BLKD
  Bổ sung Tác vụ hiện tại vào phần cuối của
  danh sách đợi và thực hiện hàm DeSchedule().

Hàm V() được thực hiện theo thuật toán sau:

- Nếu Bộ đếm ≥0 (tức là không còn tác vụ đợi)

Tăng bộ đếm (tăng số lượng tài nguyên dư)

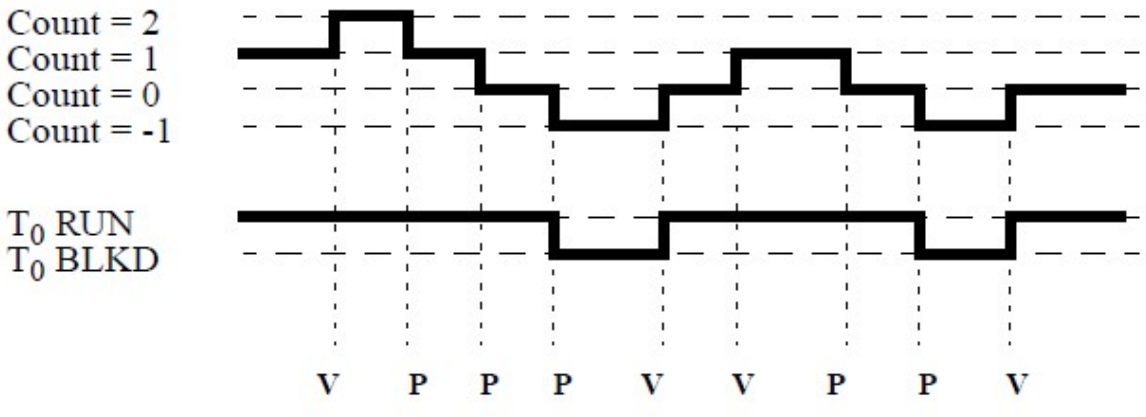
- Else

  Tăng bộ đếm (Giảm số lượng tác vụ đợi)

Gán trạng thái (State) của
Tác vụ đầu tiên trong hàng đợi về trạng thái RUN
Xóa Tác vụ đầu tiên trong hàng đợi

Hàm V() kiểm tra giá trị Bộ đếm, Nếu V() xác định bộ đếm là ≥0, có nghĩa là không có tác vụ đợi thì đơn giản chỉ tăng Bộ đếm, thông báo là có ít nhất tài nguyên dư. Nếu V() xác định bộ đếm <0, tức là có tác vụ đợi. Số lượng tác vụ đợi sẽ giảm 1 đơn vị nhờ bộ đếm tăng 1 giá trị, tác vụ đầu tiên trong hàng đợi sẽ đuwọc mở khóa và chuyển sang trạng thái chạy RUN. Đồng thời tác vụ này cũng được xóa khỏi danh sách đợi.

Hình dưới thể hiện sự hoạt động của hàm P() và V():




Một Semaphore hoạt động rất giống 1 tài khoản ngân hàng. Ở đây không hạn hạn chế số tiền gửi vào tài khoản (hàm V()). Chỉ có điều bạn có thể rút tiền ra (hàm

P()) chi khi bạn còn dư tiền trong tài khoản. Trong trường hợp này bạn cần đợi cho đến khi có ai đó cấp đủ tiền vào tài khoản. Nếu bạn cố gắng gian lận bằng cách rút tiền từ tài khoản rỗng (dùng hàm P() khi Bộ đếm bằng 0) thì bạn sẽ bị phạt tù (bị khóa) cho đến khi tiền đã hoàn trong tài khoản.

Cùng giống như tài khoản ngân hàng, có sự khác nhau rõ ràng giữa hàm P() và hàm V() như bảng dưới đây:

| P() | V() |
|---|---|
| Hàm P() không được gọi trong hàm dịch vụ ngắt ISR | Hàm V() có thể được gọi từ bất kỳ đâu, kể cả hàm dịch vụ ngắt ISR |
| Khi một hàm P() được gọi có thể khóa tác vụ | Khi một hàm V() được gọi có thể không khóa bất kỳ tác vụ nào |
| Giá trị âm của bộ đếm được giới hạn bởi số lượng tác vụ tồn tại, sau đó mỗi tác vụ được khóa khi hàm P() được gọi khi bộ đếm ≤0 | Hàm V() tác động với bất cứ giá trị nào của bộ đếm, nó chỉ tăng giá trị này lên giá trị cao hơn |
| Hàm P() đòi hỏi thời gian thời gian đáp ứng là O(N) khi bộ đếm <0 và O(1) khi ≥0. | Hàm V() thực hiện với thời gian không đổi. |

Các Semaphore sử dụng giá trị khởi tạo ban đầu phụ thuộc nào điều kiện cụ thể như bảng sau:

| Giá trị khởi tạo bộ đếm | Điều kiện |
|---|---|
| N>1 | Semaphore thể hiện còn dư N tài nguyên |
| N=1 | Có 1 tài nguyên dư cho 1 tác vụ sử dụng, ví dụ như khối phần cứng |
| N=0 | Có một hoặc một vài tài nguyên, nhưng chưa cáp phát. Ví dụ như bộ đệm để |

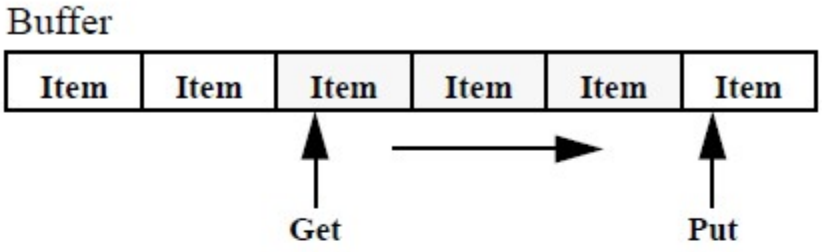
| | |
|---|---|
| | nhận dữ liệu |

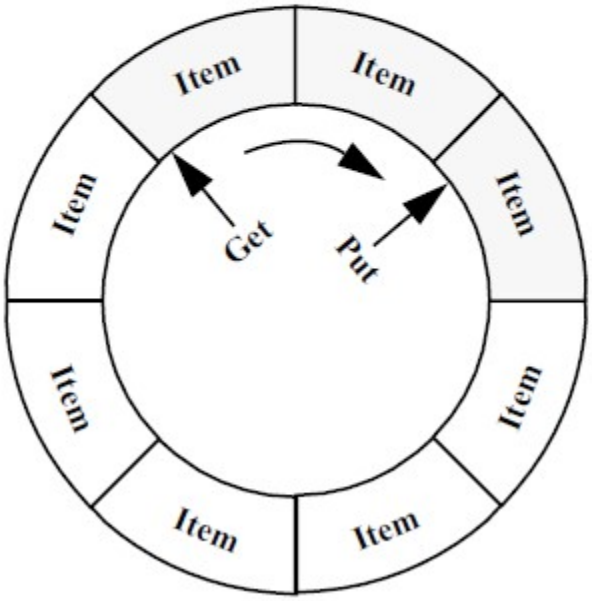
## 9. Hàng đợi

Như đã thấy ở trên thông qua Semaphore cung cấp một công cụ mạnh mẽ để điều hành chế độ đa nhiệm. Tuy nhiên trong phần này ta xem xét một cấu trúc dữ liệu hay sử dụng trong hệ điều hành thời gian thực đó là hàng đợi. Hàng đợi, là việc theo nguyên tắc FIFO, là bộ đệm dữ liệu có 2 hàm làm việc chính là Put() và Get(). Kích thước của phần tử (Item) lưu trong hàng đợi có thể biến đổi, đồng thời số lượng phần tử cũng co thể thay đổi.

**Bộ đệm vòng tròn (Ring Buffer)**

Một dạng đơn giản nhất của hàng đợi là vùng đệm vòng tròn. Một vùng nhớ liên tiếp được cấp phát cho bộ đệm, có 2 biến **GetIndex** và **PutIndex** ban đầu có giá trị 0 chỉ vào vị trí đầu tiên của vùng nhớ. Đặc điểm của 2 giá trị này là chỉ tăng, khi chỉ đến vị trí cuối vùng nhớ sẽ bị xóa về giá trị bắt đầu. Bộ đệm được coi là rỗng nếu và chỉ nếu **GetIndex= PutIndex.** Ngoài ra **PutIndex** luôn ở phía trước **GetIndex**, có trường hợp **PutIndex** nhỏ hơn GetIndex khi PutIndex qua về giá trị ban đầu khi vượt quá giá trị lơn nhất của vùng nhớ.




Thuật toán của hàm Put() để đưa một phần tử (Item) vào bộ đệm vòng như sau:

- Đợi trong trường bộ đệm bị đầy hoặc báo lỗi trong trường hợp bộ đệm bị tràn.
- Buffer[PutIndex]=Item
- PutIndex=PutIndex+1

Hàm Get() có chức năng lấy phần từ (Item) ra khỏi bộ đệm, hàm này thực hiện thuật toán sau:

- Đợi trong trường hợp bộ đệm rỗng
- Item=Bufer[GetIndex]
- GetIndex=GetIndex+1

Trong thực tế bộ đệm rỗng thường xảy ra hơn trường hợp bộ đếm tràn. Cùng giống như tài khoản của ngân hàng thường trong tài khoản không có tiền nhiều hơn là trường hợp quá nhiều tiền.

**Bộ đệm vòng tròn với Get Semaphore**

Với ý tưởng xem xét phần tử (Item) trong bộ đệm như một tài nguyên. Ta sẽ bổ sung việc thao tác với GetIndex và PutIndex một Semaphore gọi là GetSemaphore để biểu thị sự hiện diện của phần tử bên trong bộ đệm. Cũng tương tự như GetIndex và PutIndex được khởi tạo với giá trị 0, Semaphore này cũng có giá trị bộ đếm khởi tạo bằng 0.

Với mỗi hàm Put(), một hàm V() được gọi sau khi phần được đưa vào bộ đệm.

- Đợi trong trường bộ đệm bị đầy hoặc báo lỗi trong trường hợp bộ đệm bị tràn.
- Buffer[PutIndex]=Item
- PutIndex=PutIndex+1
- Gọi hàm V() cho GetSemaphore.

Với mỗi hàm Get(), một hàm P() được gọi trước khi xóa phần tử khỏi bộ đệm. Nếu trong bộ đệm không còn phần tử nào thì hàm P() bị khóa cho đến khi sử dụng hàm Put() và V() để đưa phần tử vào bộ đệm

- Gọi hàm P() cho GetSemaphore
- Item=Bufer[GetIndex]
- GetIndex=GetIndex+1

**Bộ đệm vòng tròn với Put Semaphore**

Thay cho trường hợp xem phần tử là tài nguyên, ta xem xét vùng nhớ trống trong bộ đệm là nguồn tài nguyên. Ta sẽ bổ sung việc thao tác với GetIndex và PutIndex một Semaphore gọi là PutSemaphore để biểu thị sự hiện diện vùng không gian trống trong bộ đệm. Cũng tương tự như GetIndex và PutIndex được khởi tạo với giá trị 0, Semaphore này cũng có giá trị bộ đếm khởi tạo bằng kích thước của bộ đệm (Buffer Size).

Với mỗi hàm Put() , một hàm P() thực hiện trước khi phần tử được đưa vào bộ đệm và vùng trống của bộ đệm bị giảm đi. Nếu không váo vùng trống trong bộ đệm thì hàm P() bị khóa cho đến khi gọi hàm V() để giải phóng vùng nhớ trong bộ đệm.

- Gọi hàm P() cho PutSemaphore
- Bufer[PutIndex]=Item
- PutIndex=PutIndex+1

Với mỗi hàm Get(), mọt hàm V() thực hiện sau khi xóa 1 phần tử khởi bộ đệm và báo ràng có thểm vị trí trống trong bộ đệm.

- Đợi nếu bộ đệm rỗng
- Item=Bufer[GetIndex]
- GetIndex=GetIndex+1
- Gọi hàm V() cho PutSemaphore

Sơ đồ này được sử dụng ít hơn trường hợp GetSemaphore.

**Bộ đệm vòng với Get và Put Semaphore**

Cuối cùng ta xét đến trường hợp sử dung cả hai Get và Put Semaphore với bộ đệm vòng.

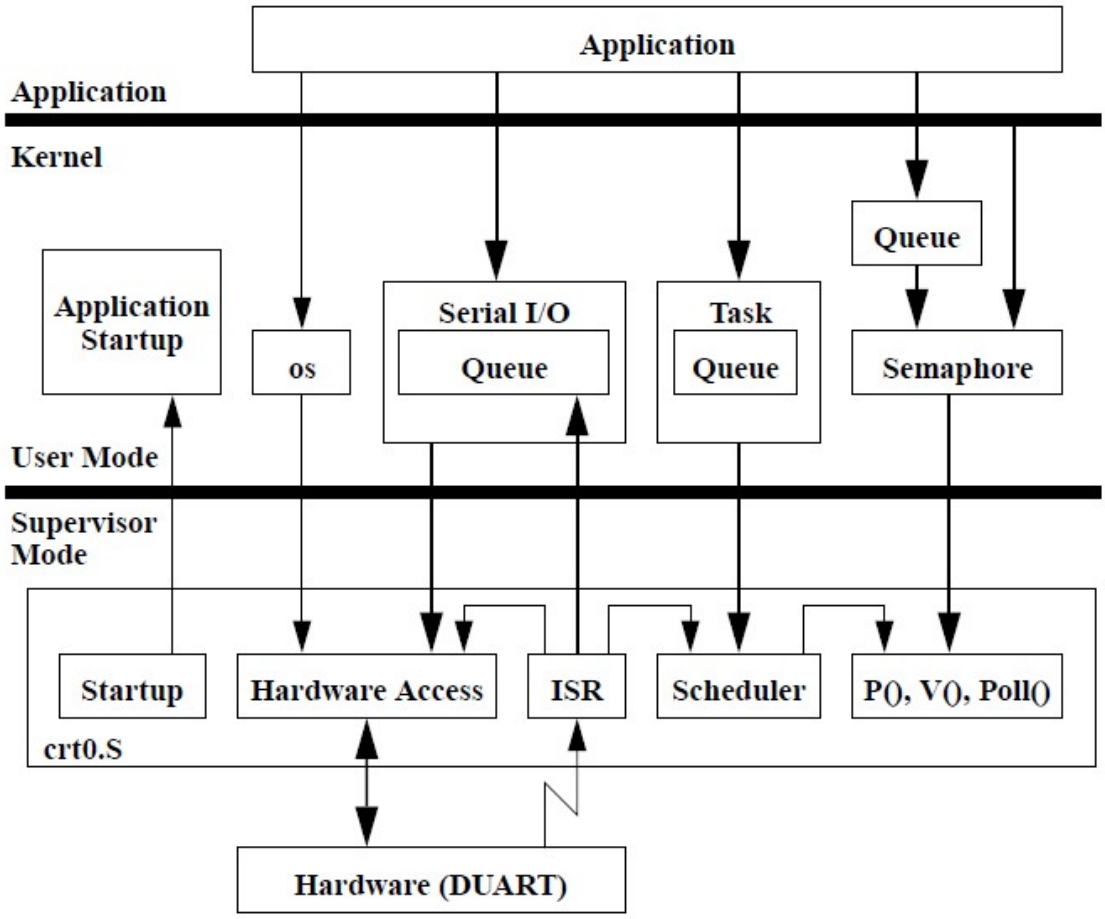
Với mỗi hàm Put(), một hàm P() thực hiện trước khi phần tử đưa vào và một hàm V() thực hiện sau khi phần tử đưa vào:

- Gọi hàm P() cho PutSemaphore
- Bufer[PutIndex]=Item
- PutIndex=PutIndex+1
- Gọi hàm V() cho GetSemaphore

Mỗi hàm Get(), một hàm V() thực hiên trước khi phần tử bị xóa và một P() thực hiện sau khi phần tử xóa từ bộ đệm.

- Gọi hàm P() cho GetSemaphore
- Item=Bufer[GetIndex]
- GetIndex=GetIndex+1
- Gọi hàm V() cho PutSemaphore.

Sơ đồ cấu trúc lõi của hệ điều hành thời gian thực



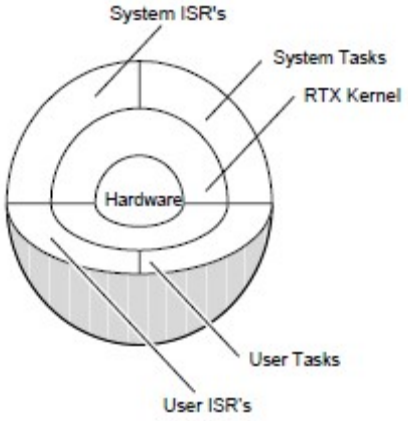

### 4.3 Hệ điều hành RTX-51.

Đối với hệ thống nhúng hệ điều hành thời gian thực RTX có hai điểm chung sau:

- Tác vụ cần thực hiện trong thời gian đủ ngắn
- Một vài tác vụ hoạt động độc lập về thời gian và thuật toán, nhưng cần hoạt động cùng lúc trên một vi xử lý.

Vấn đề đầu tiên đòi hỏi hệ thống đáp ứng tính thời gian thực.

Vấn đề thứ hai đòi hỏi cần thiết kế mô hình hệ thống có thể hoạt động nhiều chương trình hay nhiều tác vụ cùng lúc. Trong trường hợp này mỗi tác vụ riêng biệt được tổ chức như một khối tính toán độc lập.




Hệ điều hành thời gian thực RTX-51 đa nhiệm bao gồm các hàm để giải quyết 2 vấn đề trên được áp dụng cho tất cả các chip vi xử lý họ C51.

Các ưu điểm  của hệ điều khiển thời gian thực như sau:

- Một chương trình có thể được xây dựng, kiểm tra, hiệu chỉnh dễ dàng hơn. Khả năng này có được nhờ việc phân chia sự hoạt động thành các tác vụ.
- Việc module hóa cho phép các tác vụ riêng biệt được sử dụng trong các phần khác nhau của chương trình.
- Nhờ giải quyết 2 vấn đề là tính thời gian thực và chế độ đa nhiệm, nên chương trình có tính cấu trúc và dễ xử lý hơn.

Các ưu điểm của hệ điều hành thời gian thực RTX-51:

- Sử dụng đơn giản, đã tích hợp vào hệ phát triển Keil C51
- Hỗ trợ tất cả các đặc trưng của hệ c51 như tính toán dấu phẩy động, hàm chia sẻ (reentrant) và hàm ngắt.
- Sử dụng cấu hình hệ điều hành than thiện cho tất cả vi xử lý họ C51
- Khả năng mềm dẻo, chỉ sử dụng tối thiểu tài nguyên hệ thống để xây dựng hệ điều hành.

5 Kết cấu của hệ thống thời gian thực

5.1Tác vụ.

Hệ điều hành RTX-51 có 2 dạng như sau:

- Tác vụ nhanh thực hiện với thời gian ngắn và ngắt thời gian. Mỗi tác vụ nhanh có vùng thanh ghi và ngăn xếp riêng biệt. RTX-51 hỗ trợ tối đa 3 tác vụ nhanh cùng hoạt động tại 1 thời điểm.

- Tác vụ tiêu chuẩn cần nhiều thời gian hơn để thực hiện chuyển tác vụ, đồng thời sử dụng bộ nhớ bên trong ít hơn tác vụ nhanh. Tất cả các tác vụ tiêu chuẩn đều chia sẻ vùng thanh ghi và ngăn xếp. Trong khi thay đổi tác vụ nội dung hiện tại của các thanh ghi và ngăn xếp được lưu trữ trong vùng RAM ngoài. RTX-51 hỗ trợ tối đa 16 tác vụ tiêu chuẩn hoạt động cùng 1 thời điểm.

Tác vụ của RTX-51 được khai báo trong ngôn ngữ C với từ khóa "_task_"

**Tác vụ truyền thông và đồng bộ**

Trong RTX-51 cho 2 công cụ cho phép các tác vụ riêng biệt có thể truyền thông tin và đồng bộ tác vụ với nhau. Các công cụ này hoạt động độc lập tại các tác vụ khác nhau:

- Tín hiệu cho phép truyền tin nhanh nhất để phục vụ bài toán đồng bộ. Không có dữ liệu trao đổi với nhau.
- Thông báo (dữ liệu) được trao đổi giữa các tác vụ thông qua hộp thư (mailbox). Hộp thư cho phép chế độ trao đổi thông qua bộ đệm. Các tác vụ có thể nằm trong hàng đợi để nhận dữ liệu thông báo. Các thông báo riêng biệt được quản lý bởi hộp thư tuân theo nguyên tắc FIFO (Vào trước , ra trước). Nếu có vài tác vụ cùng đợi nhận thông báo thì tác vụ nào chờ lâu nhất sẽ được nhận trước.
- Semaphore là công cụ đơn giản để chia sẻ quyền sử dụng tài nguyên và chống xung đột. Bằng cách sử dụng một từ khóa biểu thị (token) của tài nguyên có thể quản lý theo nguyên tắc chỉ có 1 tác vụ tại 1 thời điểm sử dụng tài nguyên đó. Nếu có nhiều hơn 1 tác vụ mong muốn sử dụng, thì tác vụ đầu tiên sẽ được quyền truy nhập, trong khi đó tác vụ thứ 2 được đặt trong danh sách đợi cho đến khi tác vụ 1 đã thực hiện xong với nguồn tài nguyên đó.

**Chuyển tác vụ**

Hệ điều hành RTX-51 có một công cụ để điều khiển chuyển tác vụ theo nguyên tắc ưu tiên. Có một công cụ bổ sung để chuyển tác vụ là chia khoảng thời gian theo nguyên lý mạch vòng (Round-robin scheduling)

Hệ điều hành RTX-51 tổ chức thành 4 mức độ ưu tiên: mức 0,1 và 2 được sử dụng cho tác vụ tiêu chuẩn. Mức 3 được dùng cho tác vụ nhanh.

Mỗi tác vụ có thể đợi một sự kiện nào đó để tác động mà không đòi hòi đến thời gian xử lý của CPU. Các sự kiện có thể là thông báo, tín hiệu, ngắt và hiện tượng tràn thời gian hoặc là tổ hợp của sự kiện trên.

Có ba dạng đợi như sau:

- Bình thường: Tác vụ đợi (WAITING –BLOCKED) có thể được khóa với thời gian bất kỳ cho đến khi một sự kiện xuất hiện.
- Có điều kiện: Tác vụ đợi không bao giờ bị khóa, tác vụ được kích hoạt nếu có sự kiện xảy ra phù hợp với điều kiện cho trước.
- Quá thời gian (Time-out): Tác vụ bị khóa nếu trong khoảng thời gian định trước không có sự kiện phù hợp xuất hiện.

5.2 Hệ thống ngắt.

Hệ thống điều hành thời gian thực RTX-51 thực hiện tác vụ đồng bộ với các sự kiện bên ngoài như hệ thống ngắt. Có hai dạng của quá trình ngắt như sau:

- Hàm ngắt C51.
  Các ngắt được điều hành bằng hàm ngắt C51
- Tác vụ ngắt.
  Các ngắt được điều hành bằng các tác vụ nhanh và tiêu chuẩn của RTX-51

5.3Hệ thống xung nhịp đồng hồ

Trong hệ điều hành thời gian thực RTX-51 xung nhịp được xây dựng từ các phần cứng là Timer0,1 và 2 của vi xử lý. Hệ điều hành sẽ sử dụng xung nhịp này để điều hành chế độ tràn thời gian Time-out và chuyển tác vụ mạch vòng (Round-robin scheduling)

5.4Tài nguyên hoạt động.

Hệ điều hành RTX-51 yêu cầu các tài nguyên của hệ thống như sau:

- Bộ nhớ chương trình:
  Nằm trong khoảng từ 6 đến 8KB, phụ thuộc vào các hàm sử dụng
- Bộ nhớ trong RAM (DATA và IDATA):
  Từ 40 đến 46 byte cho hệ thống
  20 đến 200 byte cho ngăn xếp
  Thanh ghi bank 0 cho tác vụ tiêu chuẩn, thanh ghi bank 1,2 và 3 cho tác vụ nhanh hoặc hàm ngắt C51.

- Bộ nhớ ngoài XDATA:
  Tối thiểu 450byte
- Timer 0,1 và 2 cho hệ thống xung nhịp

6 Chương trình ví dụ

```
#pragma large

#include  <Rtx51.h>                 // RTX-51 Definitions
#define  PRODUCER NBR    0         // Tasknumber for the producer task
#define  CONSUMER NBR    1         // Tasknumber for the consumer task
#define  FIRST_MAILBOX   0         // The mailbox identification
#define  WAIT FOREVER    0xFF      // A constant, signalling to the RTX
                                   // call, that no timeout for the call is
                                   // expected.

void  ProducerTask (void)  task  PRODUCER NBR
{
   unsigned int  MessageToSend = 1;

   os_create_task (CONSUMER_NBR);   // Create the Consumer-Task
   for (;;)                         // endless loop
   {
      // Send the actual value of 'send mes' to mailbox 0
      // if the mailbox is full, wait until there is place for the message
      os_send_message (FIRST_MAILBOX, MessageToSend, WAIT_FOREVER);
      MessageToSend++;
   }
}

void  ConsumerTask (void) _task_ CONSUMER_NBR _priority_ 1
{
   unsigned int  ReceiveBuffer;

   for (;;)
   {
      // Read from mailbox FIRST_MAILBOX one message
      // Wait for a message, if the mailbox is empty
      os_wait (K_MBX + FIRST_MAILBOX, WAIT_FOREVER, &ReceiveBuffer);
      //
      // ... perform some calculations with the message
      //
   }
}

void  main (void)
{
   signed char  RtxReturnState;

   // Init the system and start the Producer-Task
   RtxReturnState = os_start_system (PRODUCER_NBR);
}
```

7 Lập trình trên hệ điều hành thời gian thực

7.1 Quản lý tác vụ

Hàm chính (main) của các tác vụ trong hệ điều hành thời gian thực thực hiện quá trình chặt trẽ về thờigian và phụ thuộc vào các sự kiện bên trong, bên ngoài. Múc độ ưu tiên có thể cài đặt cho từng tác vụ riêng rẽ phụ thuộc vào mức độ quan

trọng của nó. Trong đó giá trị 3 tương ứng với mức độ ưu tiên cao nhất và 0 với mức độ ưu tiên thấp nhất.

Hệ điều hành thời gian thực RTX -51 luôn định nghĩa tác vụ READY có mức độ ưu tiên cao nhất. Tác vụ này sẽ duy trì quyền điều khiển vi xử lý cho đến khi có một tác vụ khác có mức độ ưu tiên cao hơn sẵn sàng thực thi hoặc cho đến khi tự kết thúc hoạt động.

Nếu có một vài tác vụ READY tồn tại với mức độ ưu tiên 0, việc chuyển tác vụ có thể lựa chọn sau khi kết thúc 1 khoảng thời gian định sẵn (Round-robin scheduring)

Sử dụng chỉ dẫn sau khi cài đặt múc ưu tiên tác vụ: Chương trình cần phải hoạt động bình thường mà không phụ thuộc và mức ưu tiên tác vụ. Múc độ ưu tiên chỉ phục vụ cho việc tối ưu thời gian

7.2Trạng thái tác vụ

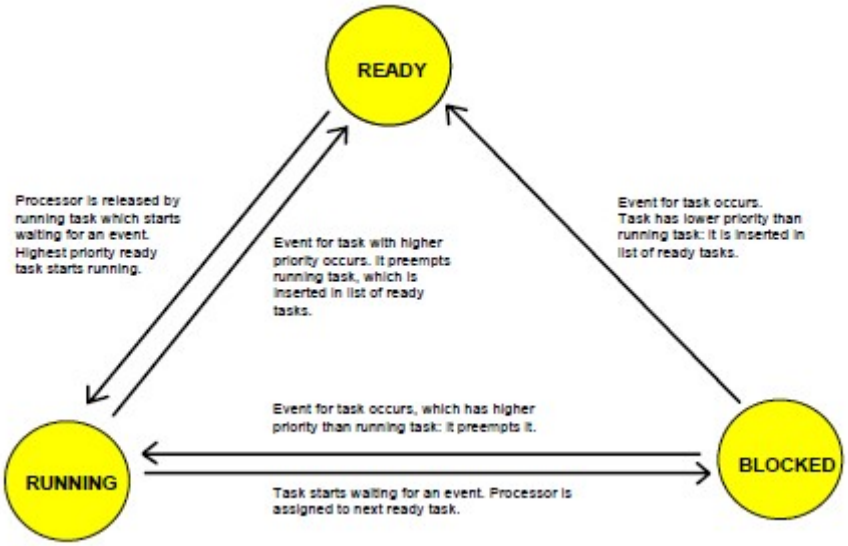
RTX-51 được tổ chức có 4 trạng thái tác vụ như sau:

| | |
|---|---|
| READY | Tất các các tác vụ có thể ở trạng thái READY. Chỉ có 1 tác vụ ở trạng thái chạy-RUNNING (ACTIVE) |
| RUNNING (ACTIVE) | Tác vụ được vi xử lý thực hiện. Chỉ có 1 tác vụ có thể ở trạng thái này tại 1 thời điểm |
| BLOCKED (WAITING) | Tác vụ đợi một sự kiện |
| SLEEPING | Tất cả các tác vụ không ở trạng thái hoạt động hoặc bị dừng |

Một sự kiện có thể xuất hiện theo chu kỳ thời gian, khi nhận 1 thông báo hoặc tín hiệu, hoặc phát sinh 1 ngắt. Các dạng sự kiện này có thể làm thay đổi trạng thái của tác vụ, hay nói một các khác nó có tác động làm chuyển tác vụ.

Các trạng thái “READY”, “RUNNING” và “BLOCKED” được gọi là các trạng thái tác vụ tích cực. Trạng thái này có được chỉ sau khi thực hiện hàm khởi tạo tác vụ “os_creat_task”. Trạng thái “SLEEPING” là trạng thái không tích cực của tác vụ, trạng thái này xảy ra khi tác vụ đã khai báo nhừng chưa được khởi tạo.

Hình dưới mô tả trạng thái của tác vụ.




7.3Chuyển tác vụ

Hệ điều hành RTX-51 khi thực hiện chương trình khai báo các tác vụ riêng biệt theo nguyên tắc lập lịch (người điều vận).

Việc lập lịch hoạt động theo luật sau:

- Tác vụ với mức ưu tiên cao nhất trong tất cả các tác vụ ở trạng thái READY sẽ được thực hiện
- Nếu có một vài tác vụ có cùng mức ưu tiên ở trạng thái READY, thì tác vụ nào có thời gian đợi lâu nhất sẽ được thực hiện.
- Chuyển tác vụ chỉ được thực hiện  nếu luật số 1 bị vi phạm ( trừ trường hợp Round-robin scheduring)

Các luật trên được tuân thủ chặt trẽ và khhong bao giừo bị vi phạm trong mọi trường hợp. Trước khi 1 tác vụ có thể chuyển trạng thái, hệ điều hành sẽ kiểm tra có phù hợp với luật không. Việc chia thời gian chuyển tác vụ (Round-robin scheduring) được thực hiện nếu các điều kiến sau được thỏa mãn:

- Round-robin scheduring cần được cho phép
- Tác vụ RUNNING cho mưc sđộ ưu tiên bằng 0 và hiện tại không thực hiện phép toán dấu phẩy động.
- Ít nhất có 1 tác vụ có mức ưu tiên 0 nằm ở trạng thái READY.

- Việc thay đổi tác vụ cuối cùng được thực hiện sau khi lựa chọn giãn cách thời gian của hệ thống ( xem hàm :os_set_slice"). Giãn cách thời gian của hệ thống có thể thay đổi động trong khi chương trình hoạt động.

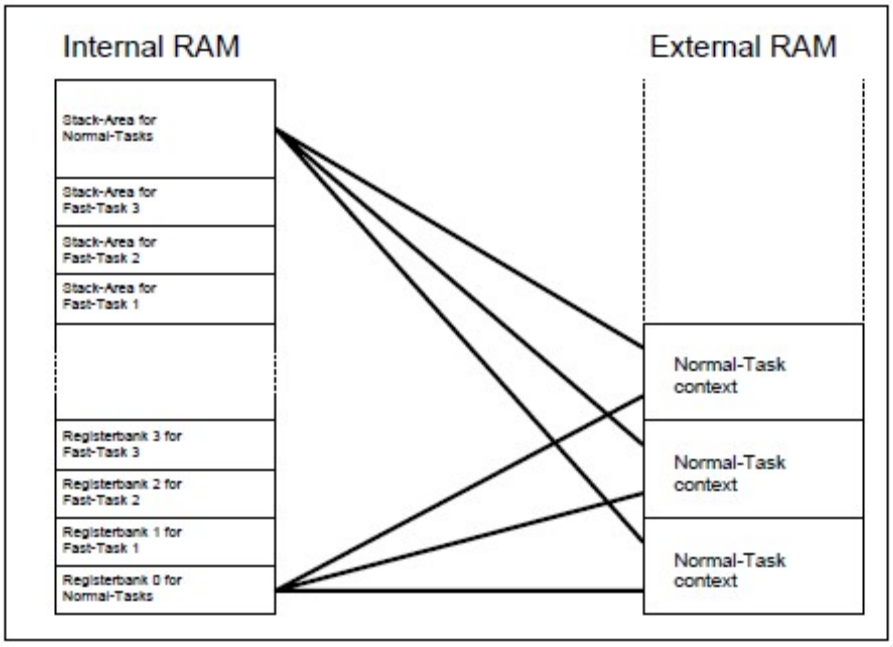
Chế độ hoạt động được ưu tiên bởi RTX-51 là chế độ đa nhiêm ưu tiên. Nếu được dừng bởi người sử dụng, tác vụ với mức độ ưu tiên bằng 0 có thể đuwọc quản lý bằng Round-robin scheduring.

7.4Các lớp của tác vụ.

Hệ điều hành RTX-51 về cơ bản gồm các lớp tác vụ sau:

**Tác vụ nhanh**

- Bao gồm đáp ứng đặc biệt ngắn và cấm ngắt thời gian
- Bao gồm một bank thanh ghi tách rời và một ngăn xếp tách rời (các bank thanh ghi 1,2 và 3)
- Bao gồm tác vụ có mức ưu tiên cao nhất (mức 3) và có thể ngắt cưỡng bức các tác vụ tiêu chuẩn
- Có cùng mức ưu tiên và không thể ngắt lẫn nhau.
- Có thể bị ngắt bởi các hàm ngắt C51
- Có tối đa là 3 tác vụ nhanh tích cực trong hệ thống




**Tác vụ tiêu chuẩn**

- Yêu cầu nhiều thời gian chuyển tác vụ hơn so với tác vụ nhanh
- Chia sẽ bank thanh ghi chung và vùng ngăn xếp chung (bank thanh ghi 0)
- Nội dung hiện tại của thanh ghi và ngăn xếp được lưu trữ ở bộ nhớ ngoài (XDATA) khi thanh đổi tác vụ
- Có thể bị ngắt bởi tác vụ nhanh
- Có thể bị ngắt lẫn nhau
- Có thể bị ngắt bởi hàm ngắt C51
- Có tối đa 16 tác vu tiêu chuẩn có thể tích cực trong hệ thống

Mỗi tác vụ tiêu chuẩn lưu trữ nội dung trong vùng bộ nhớ ngoài. Khi thay đổi tác vụ tiêu chuẩn, ất cả các thanh ghi yêu cầu của tác vụ đang hoạt động và ngăn xếp tác vụ tiêu chuẩn được lưu trữ trong vùng nội dung. Sau đó các thanh ghi và ngăn xếp của tác vụ tiêu chuẩn đuwọc nạp trở lại từ vùng nội dung của tác vụ để bắt đồng hoạt động.

Trong trường hợp tác vụ nhanh, một tác vụ thay đổi xảy ra nhanh hơn so với tác vụ tiêu chuẩn, mỗi một tác vụ nhanh đều có bank thanh ghi và con trỏ ngăn xết tách rời. Khi thay đổi tác vụ thành tác vụ nhanh, chỉ có các bank thanh ghi tích cực và con trỏ ngăn xếp cần thay đổi.

7.5Khai báo tác vụ

Hệ lập trình C51 có các hàm mở rộng để khai báo tác vụ như sau:

void func (void) [model] _task_ [priority_ <prio>]

- Các tác vụ không trả về giá trị (void)
- Không có tham số truyền cho tác vụ (void trong bảng danh sách)
- <tastno> là một số khai báo bởi người sử dụng nằm trong khoảng từ 0 đến 255. Mỗi tác vụ cần được khai báo 1 số duy nhất. Số của tác vụ này yêu cầu trong hệ điều hành gọi hàm để định dạng 1 tác vụ. Số lượng lớn nhất các tác vụ khai báo là 256. Tuy nhiên chỉ có tối đa 19 tác vụ có thể tích cực tại 1 thời điểm.
- <prio> quy định mức ưu tiên của tác vụ. Giá trị 0 tương ứng với mức ưu tiên thấp nhất và giá trị 3 tương ứng mức ưu tiên cao nhất. Nếu mưc sưu tiến không khai báo tương ứng với mức ưu tiến của tác vụ là 0.
- Các tác vụ tiêu chuẩn có thể được biên dịch với bank thanh ghi 0 là giá trị ngườiầm định của bộ biên dịch C51. Tác vụ nhanh có thể đwọc biên dịch

cho bank thanh ghi 1, 2 hoặc 3. Cần định hướng việc này thông qua từ khóa #pragma REGISTERBANK(x) (trong đó x có thể là 1,2,3). Nếu quy luật này bị vi phạm bộ biên dịch và liên kết 51 sẽ báo lỗi.

Ví dụ 1: Tác vụ tiêu chuẩn với tác vụ số 8 và mức ưu tiên 0

void example_1 (void) _tast_ 8 _priority_ 0

hoặc

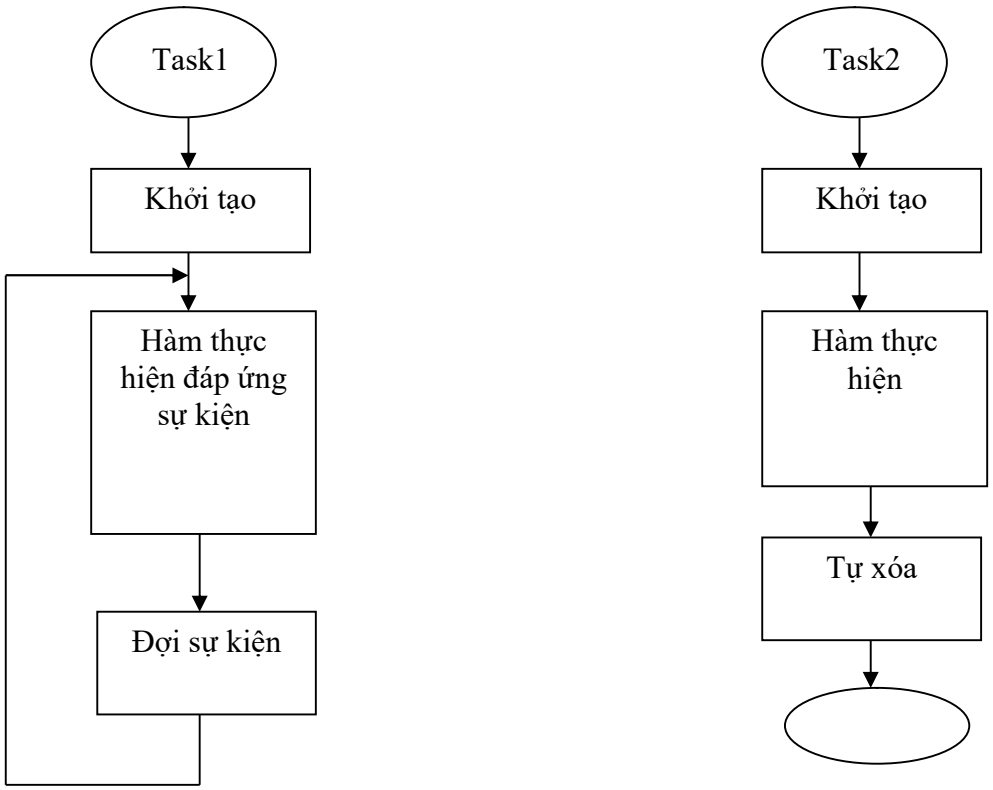
void example_1 (void) _tast_ 8

Ví dụ 2: Tác vụ nhanh với tác vụ số 134 và bank thanh ghi 1

#pragma REGISTEBANK (1)

void example_2 (void) _tast_ 134 _priority_ 3

Hai ví dụ trên được thể hiện bởi sơ đồ dưới



Tác vụ 1 ở hình trên thực hiện nhiệm vụ mỗi lần có 1 sự kiện xuất hiện. Sự kiện này xuất hiện khi nhận được 1 thông báo (message), 1 tín hiệu hoặc thời gian tràn (time-out), hoặc tổ hợp của các sự kiện trên.

Sau khi thực hiện xong, tác vụ sẽ vào trạng thái đợi sự kiện tiếp theo.

Tác vụ 2 ở hình trên chỉ thực hiện nhiệm vụ được quy định trước. Nó sẽ bị xóa sau khi thực hiện xong nhiệm vụ. Thông thường tác vụ này được thực hiện khi tự kiểm tra hệ thống trong quá trình khởi động.

7.6Quản lý ngắt

Việc quản lý và thực hiện ngắt là một nhiệm vụ quan trọng của hệ điều hành thời gian thực. RTX-51 có một vài dạng thực hiện ngắt, nó phụ thuộc vào từng bài toán ứng dụng.

**Phương pháp thực hiện ngắt.**

Hệ điều hành RTX-51 có 2 phương pháp thực hiện ngắt gồm:

(1) Hàm ngắt C51

(2) Ngắt tác vụ RTX-51:

- Ngắt tác vụ nhanh
- Ngắt tác vụ tiêu chuẩn

Phương pháp (1) đáp ứng các hàm ngắt tiêu chuẩn của vi xử lý C51với các sự kiện không liên quan đến RTX-51. Khai xuất hiện ngắt, chương trình sẽ nhẩy trực tiếp đến hàm ngắttương ứng, quá trình này sẽ độc lập với tác vụ hiện tại đang thực hiện. Ngắt là quá trình thực hiện bên ngoài RTX-51 và độc lấp với luật thực hiện danh sách tác vụ.

Phương pháp (2), một tác vụ nhanh hoặc tác vụ tiêu chuẩn đuwọc sử dụng để thực hiện ngắt. Trước khi ngắt xuất hiện, tác vụ đợi (WAITING) được chuyển thành READY và bắt đầu hoạt động phù hợp với luật thực hiện dach sách tác vụ. Quá trình thực hiện đuwọc tích hợp bên trong hệ điều hành RTX-51. Một ngắt cứng dược gọi khi nhận được thông báo hay tín hiệu.

Các ngắt dạng này có những ưu nhược điểm như sau:

| Phương pháp | Ngắt C51 | Ngắt nhanh | Ngắt tiêu chuẩn |
|---|---|---|---|
| Thời gian đáp ứng ngắt | Rất nhanh | nhanh | Chậm |
| Quá trình cấp ngắt | Rất ngắn, dùng | Hàm hệ thống đặc | Tất cả các hàm của |

| | | | |
|---|---|---|---|
| | hàm hệ thống | biệt, các ngắt nhanh khác | hệ thống, tác tác vụ nhanh |
| Bảng ngắt | - | Hàm dịch vụ ngắt | Hàm dịch vụ ngắt, tác vụ nhanh, tác vụ tiêu chuẩn với mức ưu tiên cao |
| Sử dụng tài nguyên | Nhiều (ngăn xếp và các bank thanh ghi) | Nhiều (ngăn xếp và các bank thanh ghi) | Một vài ( ngăn xếp và bank thanh ghi đươch chia sẽ với các tác vụ tiêu chuẩn khác) |
| Dạng ngắt | Tĩnh (chỉ 1 nguồn ngắt tao ra dịch vụ ngắt) | Động (cho phép đa nguồn ngắt với tác vụ) | Động (cho phép đa nguồn ngắt với tác vụ) |
| Cho phép hệ điều hành RTX-51 gọi | Một vài trường hợp đặc biệt | Tất cả | Tất cả |

Các đặc trưng chính của các ngắt trên như sau:

- Hàm ngắt C51

Xảy ra đột ngột hoặc có chu kỳ, đặc tính quan trọng là nó hoạt động độc lập với trạng thái hiện tại của hệ thống

- Ngắt tác vụ nhanh

Ngắt quan trọng hoặc có chu kỳ có liên kết với hệ thống khi nó xảy ra.

- Ngắt tác vụ tiêu chuẩn: hiếm khi dùng ngắt

Hệ điều hành thời gian thực có thời gian đáp ứng khác nhau với tác vụ nhanh và tiêu chuẩn.

7.7Tác động cho phép ngắt

Thanh ghi cho phép ngắt của vi xử lý C51 được quản lý bởi RTX-51 và người sử dụng không được trực tiếp thay đổi.

RTX-51 điều khiển bit cho phép ngắt theo luật sau:

- Ngắt ISR có thể ngắt tất cả các tác vụ và hàm hệ thống tại bất cứ thời điểm nào. Các ngắt ISR được cấm bởi các câu lệnh ngắn.
- Ngắt ISR có thể cấm và cho phép ngắt bởi người sử dụng nhờ 2 hàm hệ thống (os_enable_isr và os_disable_isr)
- Nguồn ngắt được gán cho một tác vụ nếu tác vụ đó thực sự đợi ngắt xuất hiện. Điều đó ngăn cản ngắt không mong muốn xuất hiện trong hệ thống.
- Nếu tác vụ hoạt động là tác vụ nhanh, tất cả các ngắt khác sẽ bị cấm (ngoại trừ ISR). Ngắt chưa quan trọng không ngắt tác vụ nhanh.
- Nếu tác vụ hoạt động là tác vụ tiêu chuẩn, nó có thể ngắt bởi tất cả các ngắt khác. Nếu tác vụ tiêu chuẩn khác đang đợi một trong số các ngắt xuất hiện, nó sẽ được chuyển sang trạng thái READY bởi hệ điều hành. Tuy nhiên, nó chỉ được hoạt động nếu có mức độ ưu tiên cao hơn tác vụ hiện đang chạy.
- Tất cả các ngắt tác vụ tiêu chuẩn được cấm khi thực hiện hàm hệ thống
- Ngắt thời gian hệ thống (phần cứng Timer 0 hoặc 1) hoạt động tương tự như ngắt tác vụ nhanh.

7.8Hoạt động của thanh ghi ưu tiên ngắt.

Thanh ghi ưu tiên ngắt của vi xử lý C51 không tác động đến hệ điều hành RTX-51. Khi hệ thống đang hoạt động bình thường, hệ điều hành RTX đảm bảo rằng các ngắt vẫn sẽ làm việc theo cài đặt của người sử dụng.

Tất cả các ngắt tác vụ của RTX làm việc giống như ưu tiên ngắt phần cứng. Tuy nhiên quá trình ngắt ISR chưa chắc được tối ưu nếu ngắt tác vụ và ISR được cài đặt cùng mức độ ưu tiên.

7.9Khai báo hàm ngắt C51

Hàm ngắt có thể được khai báo như sau:

void func(void) [model] [reentrant] interrupt n [using n]

- Khi các hàm ngắt được sử dụng, bank thanh ghi sử dụng có thể giống hoặc khác nhau
- Với việc dùng bank thanh ghi:

  Khi vào ngắt, hàm ngắt sẽ lưu trữ các thanh ghi ACC,B,DPH,DPL và PSW vào trong ngăn xếp của tác vụ ngắt. Trước khi các thanh ghi được ghi, người sử dụng phải chắc chắn rằng hàm ngắt không sử dụng bank thanh ghi mà hệ điều hành RTX sử dụng. Bank thanh ghi 0 cũng không được sử dụng ( do

luôn được tác vụ tiêu chuẩn và hệ thông xung nhịp dùng) Các bank thanh ghi 1,2 hoặc 3 có thể được sử dụng nếu không dùng đến tác vụ nhanh.

- Không dùng bank thanh ghi

Khi không dùng bank thanh ghi, tất các các thanh ghi yêu cần sẽ được ghi và ngăn xếp. Thao tác này sẽ mát nhiều thời gian hơn và làm tăng kích thước ngăn xếp. Trong trường hợp này các bank thanh ghi được hệ điều hành RTX thoải mái sử dụng.

- Các hàm ngắt C51 khi sử dụng bank thanh ghi tuyệt đối không sử dụng bank 0 và một trong những bank mà tác vụ nhanh sử dụng.

7.10 Tác vụ truyền thông.

Các tác vụ riêng biệt với hệ điều hành thời gian thực có thể hoạt động độc lập. Chúng có thể sử dụng chung dữ liệu, trao đổi thông tin với nhau hoặc phối hợp thực hiện nhiệm vụ chung.

Hệ điều hành RTX bao gồm 2 công cụ truyền thông tin là hộp thư mailbox và gửi tín hiệu (signal).

**Tín hiệu (signal)**

Tín hiệu là dạng đơn giản và nhanh nhất đẻ truyền tin của tác vụ. Nó hay được sử dụng do việc đồng bộ dễ dàng và không yêu cầu trao đổi dữ liệu.

Mỗi tác vụ tích cực có các cờ tín hiệu có thể thực hiện tác thao tác sau:

- Đợi một tín hiệu
- Gửi tín hiệu
- Xóa tín hiệu

Số hiệu tác vụ của tác vụ nhận dùng để nhận dạng tín hiệu cho mỗi tác động cụ thể.

**Đợi tín hiệu**

Mỗi tác vụ có thể đợi cờ tín hiệu (hàm hệ thống là “os_wait”). Nó đợi cho đến khi cờ tín hiệu này được dựng lên bởi môth tác vụ khác (hàm hệ thống là “os_send_signal”). Sau khi 1 tín hiệu đã nhận được, Tác vụ đợi xóa tín hiệu cờ và chuyển sạng trạng thái READY hoặc RUNNING phục thuộc vào quan hệ mức ưu tiên.

Nếu tín hiệu cờ đã được dựng lên khi một tác vụ gọi hàm đợi (tức là khi tín hiệu cờ đã được dựng lên với tác vụ khác) thì nó sẽ thực hiện ngay việc nhận tín hiệu. Tác vụ này không thực hiện ở trạng thái đợi (WAIT- BLOCKED).

Thời gian đợi có thể bị hạn chế. Nếu thời gian quy định bị vượt quá mà không nhận được tín hiệu thì tác vụ đợi chuyển sang trạng thái READY với một thông báo lỗi là tràn thời gian (time-out)

**Gửi tín hiệu**

Mỗi tác vụ và mỗi hàm ngắt có thể dựng cờ tín hiệu của tác vụ khác bất kỳ (gửi 1 tín hiệu cho tác vụ này). Chi 1 tín hiệu được gửi và lưu trữ với tác vụ. Nếu quá lâu mà bên nhận không nhận được, xác nhận việc gửi lỗi.

**Xóa tín hiệu**

Một tác vụ có thể xóa tín hiệu cờ của tác vụ khác bất kỳ. Việc này cho phép thay đổi trạng thái tín hiệu trong hệ thông ở bất kỳ thời điểm nào.

**Hộp thư (Mailbox)**

Với hộp thư, các thông báo (massege) với thể được trao đổi mộ cách an toàn giữa các tác vụ cụ thể.

Hệ điều hành RTX-51 cung cấp các mã hiệu cố định cho 8 hộp thư. Các thông báo có thể được trao đổi từ (word -2 byte) giữa các hộp thư này. Trong trường hợp này, một thông báo có thể là dữ liệu thực tế được truyền hoặc là mã nhận dạng của bộ đệm dữ liệu (việc này được định nghĩa bởi người sử dụng). So sánh với tín hiệu, hộp thư không dành cho tác vụ cụ thể, nhưng có thể được sử dụng bởi tất các các tác vụ và hàm ngắt. Việc tuye nhập hộp thư được thực hiện thống qua mã hiệu hộp thư.

Hộp thư cho phép các thao tác sau:

- Gửi thông báo
- Đọc thông báo

**Danh sách hộp thư**

Mỗi một hộp thư bao gồm bên trong 3 danh sách đợi. Người sử dụng không thể trực tiêp truy nhập và danh sách này. Năm bắt đuwọc các chức năng này mới hiểu đuwọc hoạt động của hộp thư.

Danh sách đợi bao gồm các trạng thái hoạt động sau:

| Mô tả trạng thái | Danh sách thông báo | Ghi danh sách đợi | Đọc danh sách đợi |
|---|---|---|---|
| Không có thông báo | Rỗng | Rỗng | Rỗng |

| | | | |
|---|---|---|---|
| Không có tác vụ đợi | | | |
| Không có thông báo<br>Có tác vụ muốn đọc thông báo | Rỗng | Rỗng | Không rỗng |
| Có các thông báo<br>Không có tác vụ đợi | Không rỗng | Rỗng | Rỗng |
| Danh sách thông báo đầy, Có tác vụ muốn ghi thông báo | Đầy | Không rỗng | Rỗng |

Ba danh sách trên có các hàm thực hiên sau:

(1) Danh sách thông báo: Liệt kê các thông báo đã được ghi vào hộp thư. Tối đa có 8 thông báo

(2) Danh sách đợi ghi: Danh sách đợi các tác vụ muốn ghi 1 thông báo vào trong danh sách thông báo của hộp thư (tối đa là có 16 tác vụ)

(3) Danh sách đợi đọc: Danh sách đợi các tác vụ muốn đọc 1 thông báo từ danh sách thông báo của hộp thư (tối đa là có 16 tác vụ)

Tất cả 3 danh sách đợi trên đều áp dụng nguyên tắc FIFO (Vào trước , ra trước) không có thứ tự ưu tiên. Tức là khi đọc, tác vụ nào chời lâu nhất (đầu tiên trong hàng đợi) sẽ lấy thông báo cũ nhất trong hộp thư.

**Gửi 1 thông báo vào hộp thư**

Mỗi tác vụ có thể gửi 1 thông báo vào một hộp thư bất kỳ. Trong trường hợp này, trong trường hợp này thông báo gửi sẽ được copy vào trong danh sách thông báo. Tác vụ gửi sau đó được tự do truy nhập để thông báo đó.

Nếu danh sách thông báo đã đầy trong quá trình gửi, tác vụ sẽ chuyển sang chế độ đợi (Vào trong danh sách đợi ghi). Nó sẽ ở trạng thái này cho đến khi có một tác vụ khác đã lấy 1 thông báo từ hộp thư và giải phóng chỗ. Tuy nhiên hệ thống sẽ giới hạn thời gian để gửi thông báo.

Nếu danh sách thông báo không bị đầy, khi xuất hiên quá trình gửi thông báo sẽ được copy vào danh sách thông báo và tác vụ không cần phải đợi.

**Đọc thông báo từ hộp thư**

Mỗi tác vụ có thể đọc 1 thông báo từ hộp thư bất kỳ. Nếu danh sách thông báo của hộp thư rỗng (không có thông báo), tác vụ sẽ chuyển sang trạng thái đợi (được đưa vapf danh sách đợi đọc)

Quá trình đợi sẽ tiếp tục cho đến khi một tác vụ khác gửi thông báo vào hộp thư. Tuy nhiên hệ thống sẽ giới hạn thời gian để đọc thông báo.

Nếu danh sách thông báo không rỗng, tác vụ đọc sẽ nhận thông báo và tác vụ không cần phải đợi.

**Semaphore**

Khái niệm Semaphore đuwọc hiểu là các nguồn tài nguyên cần được chia sẽ cho các tác vụ mà không có xung đột.

Trong hệ thống đa nhiệm thường xuyên các tác vụ sử dụng các ngườiồn tài nguyên của hệ thống. Khi một vài tác vụ có thể sử dụng cùng port của vùng nhớ, cùng kênh truyền thông nối tiếp hoặc nguồn tài nguyên khác, chung ta phải tìm cách loại trừ các khả năng xung đột của hệ thống. Semaphore là một cách thức  để điều hahf quyền kiểm soát các tài nguyên chia sẻ

Semaphore chứa một chỉ thị -khóa (token) là chương trình của bạn được quyền tiếp tục thực hiện. Nếu nguồn tài nguyến đó đã được sử dụng thì sẽ yêu cầu tác vụ dừng (khóa) cho đến khi khóa được trá lại cho Semaphore.

Có 2 dạng Semaphore là Semaphore nhị phân và Semaphore bộ đếm.  Tương ứng với tên của nó: semaphore nhị phân chỉ có 2 giá trị 1 và 0 (có khóa hay không). Với Semaphore bộ đếm cho phép các giá trị từ 0 đến 65535.

Hệ điều hành RTX-51 cung cấp  mã hiệu cố định cho 8 Semaphore dạng nhị phân.

Các Semaphore cho phép các thao tác sau:

- Đợi khóa (token)
- Trả (gửi) khóa

**Đợi khóa**

Một tác vụ yêu cầu một tài nguyên được điều khiển bởi Semaphore có thể lấy khóa từ Semaphore này bằng động tác đợi (hàm os_wait”). Nếu một khóa cho phép dùng thì tác vụ sẽ tiếp tục thực hiện. Nếu không nó sẽ bị dừng cho đến khi có khóa hoặc quá thời gian chờ.

**Trả khóa**

Sau khi thực hiện xong nhiệm vụ với tài nguyên, tác vụ sẽ trae lại khóa về Semaphore bởi hàm gửi (hàm ôs_send_token")

## 7.11 Quán lý bộ nhớ động

Không gian vùng nhớ động thường được sử dụng trong hệ đa nhiệm để lưu các kết quả và thông báo trung gian. Việc yêu cầu và giải phóng khối vùng nhớ cụ thể cần được thực hiện với thợi gian cố định trong hệ điều hành thời gian thực.

Quán lý bộ nhớ sử dụng các hàm điều hành khối vùng nhớ với kích thước quy định bới các hàm C tiêu chuẩn (malloc và free)

Hệ điều hành thời gian thực sử dụng các thuật toán đơn giản và hiệu quả với các hàm làm việc với khối vùng nhớ có kích thước cố định. Tất cả các khối vùng nhớ có cùng kích thước được quản lý bằng cách cấp phát vùng nhớ (pool memory). Có tối đa 16 vùng nhớ được cấp phát có kích thước định nghĩa sẵn. Có tối đa 255 vùng nhớ nhở được quản lý trong vùng nhớ có kích thước cố định.

**Tạo vùng nhớ (pool memory)**

Người sử dụng có thể tạo ra tối đa 16 vùng nhớ pool với kích thước cho trước. Cần sử dụng bộ nhớ ngoài XDATA để làm việc này. Vùng nhớ này đuwọc ghi nhận và quản lý bới hệ điều hành RTX (hàm os_creat_pool)

**Yêu cầu khối bộ nhớ từ vùng nhớ Pool**

Sau khi bộ nhớ pool đuwọc khởi tạo, ứng dụng có thể đòi hỏi sử dụng ngay khối vùng nhớ trong pool. Trong trường hợp này cần khái báo kích thước khối vùng nhớ sử dụng.

Nếu còn khối troóng trong vùng nhớ pool, hệ điều hành bắt đầu đánh địa chỉ của khối này cho ứng dụng. Nếu không có khối trống, sẽ trả về giá trị con trỏ null ( xem hàm "os_get_block")

**Trả lại khối vùng nhớ cho bộ nhớ Pool**

Nếu ứng dụng không sử dụng khối bộ nhớ nữa, nó có thể trả lại cho vùng nhở Pool để ứng dụng khác dùng.

## 7.12 Quản lý thời gian.

Hệ điều hành RTX có một bộ đếm thời gian bên trong, dùng để đo thời gian tương ứng đã trôi qua sau khi hệ thống bắt đầu hoạt động. Việc này được thực

hiện nhờ bộ định thời (timer) của vi xử lý. Thời gian tối qua giữa các ngắt của bộ định thời gọi là nhịp thời gian của hệ thống (system time slice).

Nhịp thời gian này để phục vụ cho các dịch vụ đặc biệt như tạm dùng hay thời gian tràn (time-out) của tác vụ đợi.

Có 3 hàm thời gian tương ứng như sau:

- Cài đặt nhịp thời gian hệ thống
- Trễ một tác vụ
- Chu kỳ tích cực của tác vụ

**Đặt nhịp thời gian hệ thống**

Chu kỳ giữa các ngắt của bộ định thời hệ thống là hạt nhân của nhịp thời gian. Độ lớn của chu kỳ được gọi là thời gian cơ sở (time slice) có thể được cài đặt trong một phạm vị rộng (hàm "os_set_sclice")

**Trễ tác vụ**

Một tác vụ có thể được trễ với thời gian là số nguyên của thời gian cơ sở. Trong lúc gọi hàm này, tác vụ sẽ bị khóa (ngủ ) cho đến khi một khoảng tời gian trôi qua (hàm "os_wait).

**Chu kỳ tích cực của tác vụ**

Với nhiều ứng dụng thời gian thực, cần thực hiện tác vụ một cách có chu kỳ. Chu kỳ tích cực tác vụ có thể đặt bởii giãn cách của hàm đợi (hàm "os_wait") .

## 7.13 Các điểm đặc biệt với hệ C51

Để thích nghi với các hàm ngắt C51, hệ điều hành thời gian thực hỗ trợ bộ biên dịch đặc biệt RTX.

Bộ biên dịch có các đặc điểm sau:

**Mô hình bộ nhớ C51**

Một ứng dụng hệ điều hành thời gian thực RTX-51 co thể sử dụng tất cả các mô hình bộ nhớ như SMALL, COMPACT, LARGE). Trong đó mô hình COMPACT được sử dụng để phục vụ hàm Reentrant.

Việc lựa chọn mô hình bộ nhớ chỉ có hiệu lực trong module con của chương trình.Các biến trong hệ điều hành thời gian thực RTX luôn được câ[s pháp trong vùng bộ nhớ ngoài XDATA. Tất cả các ứng dụng RTX-51 đều đòi hỏ bộ nhớ ngoài, như vậy hệ thống không có bộ nhớ ngoài không thể dùng cho hệ điều hành thời gian thực.

Thông thường các ứng dựng RTX sử dụng mô  hình LAGRE. Các biến mà thời gian cần truy nhập nhanh có thể chọn trong vùng nhớ trong RAM.

**Hàm Reentrant**

Các hàm thông thường của hệ C51 không được sử dụng đồng thời bởi một vài tác vụ hoặc hàm ngắt. Các hàm này lưu trữ các tham số và dữ liệu cục bộ trong đoạn bộ nhớ tĩnh. Khi nhiều hàm cùng gọi có thể có hiện tượng bị ghi đè dữ liệu.

Để giải quyết vấn đề này C51 cung cấp hàm Reentrant. Trong trường hợp này, các tham số và biến dữ liệu cục bộ được bảo vệ chống chế độ gọi nhiều lần. Hệ điều hành RTX hỗ trợ việc sử dụng hàm Reentrant trong mô hình bộ nhớ COMPACT. Trong trường hợp này, các ngăn xếp riêng rẽ có kích thước phù hợp sẽ được tạo ra và quản lý cho mỗi tác vụ. Hàm ngắt  có thể sử dụng các ngăn xếp này trong ngắt tác vụ RTX.

- Hệ điều hành RTX chỉ hỗ trợ các hàm Reentrant trong mô hình bộ nhớ COMPACT.
- Mỗi tác vụ có ngăn xếp riêng biệt
- Các hàm Reentrant có thể được sử dụng kết hợp với hàm No-reentrant với mô hình bộ nhớ SMALL và LAGRE. Việc sử dụng đồng thời hàm reentrant và non-reentrant không cho phép trong mô hình bộ nhớ COMPACT.

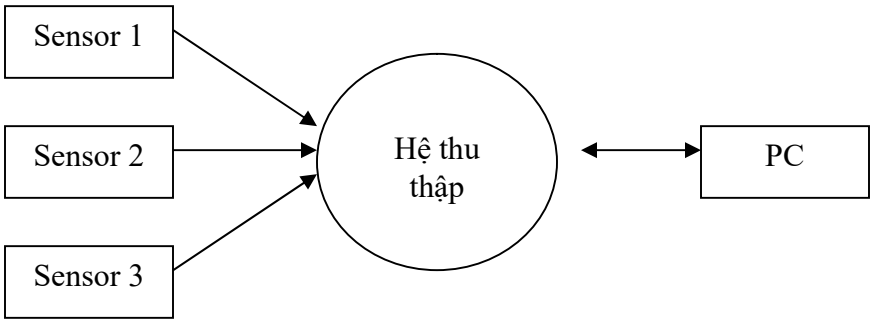
# CHƯƠNG 5. KỸ THUẬT LẬP TRÌNH CHO HỆ ĐA VI XỬ LÝ:

5.1 Khái niệm chung:

Nhiều bài toán cần sử dụng số lượng cổng port, số Timer, RAM lớn hơn nguồn tài nguyên của 1 vi xử lý sẵn có, ví dụ như cần 60 cổng port, 6 timer, 2 kênh truyền UART,... lúc này có thể sử dụng giải pháp dùng 2 vi xử lý ghép nối với nhau với giá thành tương đối thấp.

Với 1 hệ đa vi xử lý chương trình xây dựng theo module ví dụ cần chế tạo 1 đồng hồ điện tử ta có thể xây dựng 2 module: Module thứ nhất có nhiệm vụ xác định và điều chỉnh thời gian, module thứ hai có nhiệm vụ hiển thị thời gian có các dạng như LCD, Led 7 thanh hoặc động cơ bước.

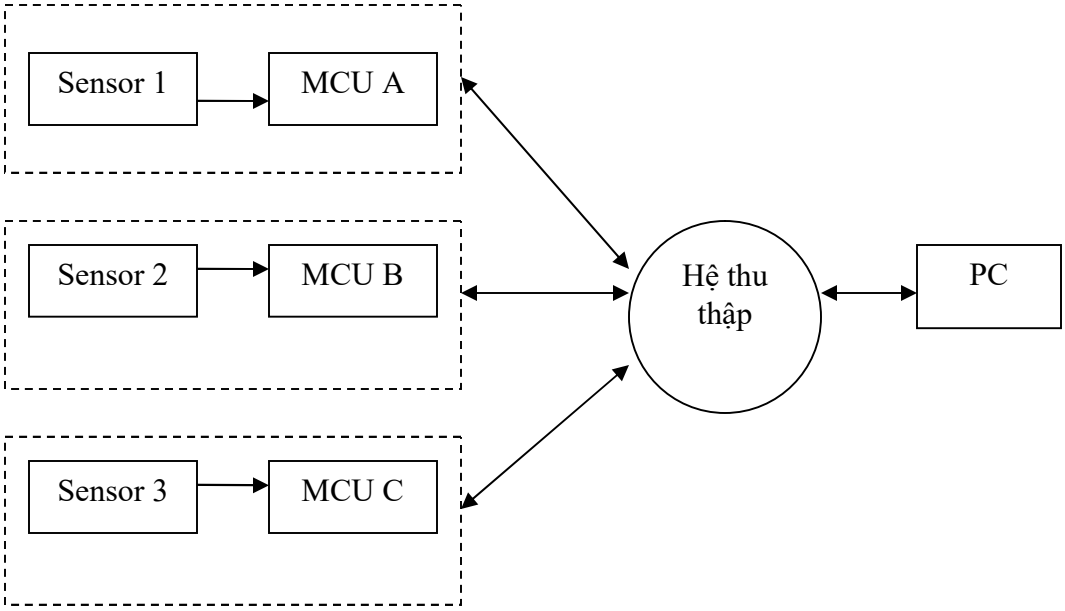
Một ví dụ khác, cần thiết kế 1 hệ thống thu thập số liệu đa kênh:




Trong trường hợp dây nối từ Sensor bị đứt, các dữ liệu từ Sensor đó bị mất, tuy nhiên hệ thu thập không xác định được lỗi do bị đứt cáp mà vẫn nhận dữ liệu sai từ sensor.

Có 1 giải pháp khác là thiết kế dạng sensor thông minh, mà khâu đo lường được thực hiện ngay tại chỗ, việc truyền dữ liệu về hệ thu thập được thiết lập thông qua 1 giao thức quy định sẵn.

Lúc này việc xác định việc đứt đường kết nối sẽ rất dễ dàng do giữa hệ thu thập và vi xử lý tại đầu đo có sự trao đổi thông tin cho nhau.

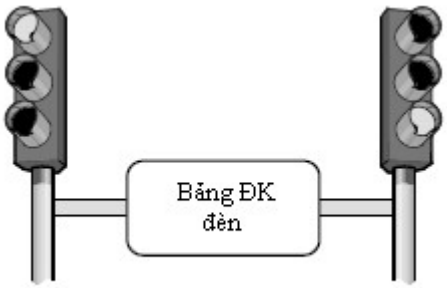



Như vậy ở đây ta vừa xây dựng đường kết nối giữa các vi xử lý trong cùng 1 hệ thống có những đặc điểm sau:

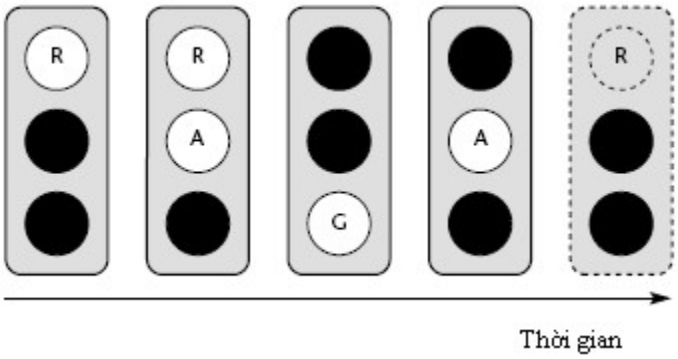
- Đồng bộ sự hoạt động của các vi xử lý
- Truyền dữ liệu giữa các vi xử lý
- Kiểm tra lỗi của các nút kết nối trong mạng

Ví dụ về đồng bộ đồng hồ giữa các điểm nút trong mạng.

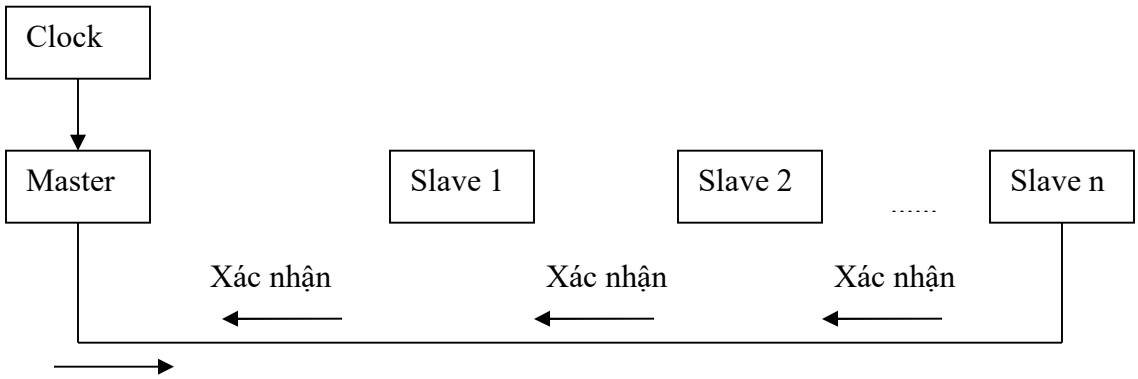
Ngoài nhiệm vụ đồng bộ đồng hồ, trong nhiều ứng dụng cần truyền dữ liệu giữa các vi xử lý trong mạch kết nối. Ví dụ trong bài toán điều khiển đèn giao thông có hỗ trợ công cụ kiểm tra cháy bóng đèn, giả sử có bóng đèn cháy thì hệ thống đèn giao thông bị nhiễu loạn dẫn đến lầm lẫn việc điều khiển phương tiện.




Việc truyền dữ liệu giữa các nút cho phép xử lý tình huống trên, ví dụ như đồng thời tắt các đèn  cùng báo lỗi hệ thống cho người sử dụng.

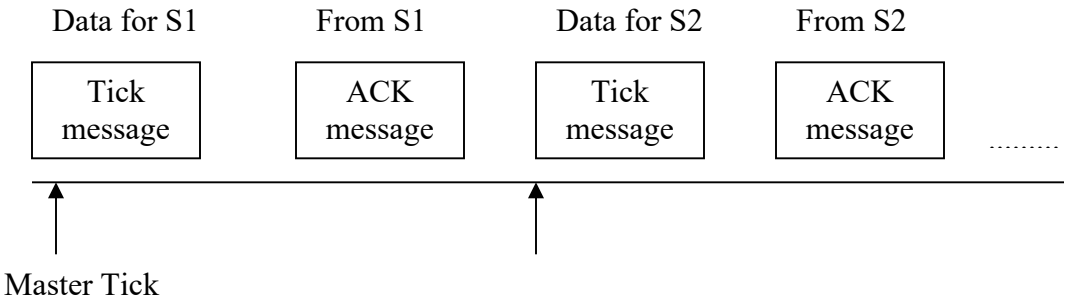



Tuần tự thời gian điều khiển đèn tín hiệu




Nhịp thông báo từ Master đến Slave

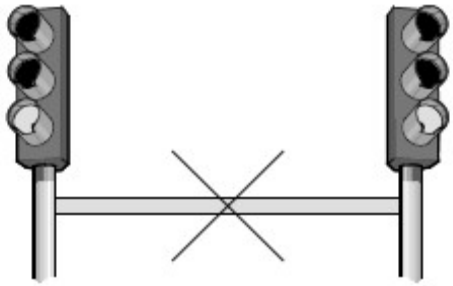
Ngoài ra việc truyền dữ liệu từ trạm Slave về Master, còn cho ta 1 khả năng xác nhận dữ liệu đồng bộ từ trạm Slave.




Cuối cùng ta xem xét khả năng kiểm tra và xác định lỗi trong các trường hợp :

- Lỗi truyền thông bao gồm lỗi do cáp, thiết bị thu phát, đầu nối,...

- Lỗi tại trạm chủ Master
- Lỗi tại trạm trở Slave



Hệ thống điều khiển đèn có sự cố do đứt đường truyền

Cơ chế kiểm tra tại trạm chia như sau:
Trạm chủ vẫn phải nhận được thông báo xác nhận (ACK message) từ mỗi trạm trở (Slave) tương ứng với khoảng thời gian nhất định, ví dụ quá trình kiểm tra của 1 trạm chủ và 10 trạm trở như sau:

- Trạm chủ giữ 1 thông báo đồng bộ cho tất cả các nút trong mạng với chu kỳ ví dụ là 1ms, thông báo cho các trạm trở với chu kỳ 10ms (các lệnh hoặc các dữ liệu cho trạm trở riêng biệt).
- Mỗi trạm trở giữ 1 thông báo xác nhận về trạm chủ chỉ khi nhận được dữ liệu tương ứng với nó (ID).

Cơ chế kiểm tra lỗi tại trạm tớ:
Tương ứng với 1 chu kỳ nhất định tại trạm trở vốn phải nhận được thông báo từ trạm chủ nếu không có sẽ chuyển sang chương trình xử lý hồi.
Cơ chế xử lý lỗi tại trạm tớ.
Khi phát hiện ra 1 lỗi, tại trạm trở vẫn xử lý như sau:

- Trạm trở sẽ khởi động lại
- Trạm trở sẽ ở trạng thái nhận cho đến khi nhận được lệnh “Start” từ trạm chủ.

Cơ chế xử lý lỗi tại trạm chủ.

- Chuyển sang trạng thái bảo vệ và dừng truyền dữ liệu lên mạng: đây là giải pháp đơn giản, việc dừng truyền dữ liệu từ master sẽ làm cho tất cả các nút Slave khởi động lại và nó sẽ chuyển sang trạng thái nhận dữ liệu. Việc truyền dữ liệu được khôi phục khi tại trạm Slave khởi động lại xong.

- Khởi động lại mạng bằng cách khởi động lại trạm master.
- Khi một trạm trở bị lỗi , thay bằng khởi mạng ta sẽ chạy một khối dự phòng.

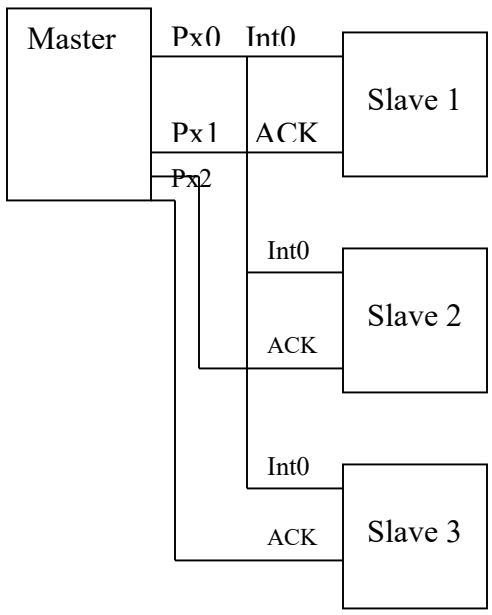
2. Phương pháp đồng bộ thời gian sử dụng ngắt ngoài.

Trạm Master sẽ tạo ra 1 xung đồng bộ được nối vào ngắt ngoài của trạm trở Slave. Phương pháp này khá hiệu quả và đơn giản giảm kích thước bộ nhớ, phần cứng phụ trợ và không sử dụng nhiều tài nguyên của CPU. Một số đặc điểm của dạng này:

- Dùng chương trình đơn nhiệm
- Mạng bao gồm 1 Master và n Slave ($n \geq 1$)
- Master cần sử dụng mạnh dao động chính xác cao, và với một chu kỳ lần lượt giữ cho các Slave các bản thông báo.
- Các Slave sẽ hoạt động căn cứ vào ngắt tác động từ Master và không sử dụng Timer bên trong để điều hành sự hoạt động của các tác nghiệp trong bản danh sách.
- Mỗi 1 Slave có 1 WatchDog Timer để kiểm tra lỗi từ Master

Ngoài ra, đa số hệ thống sử dụng nguyên tắc kết nối dùng ngắt có đặc điểm sau:

- Trạm chủ sẽ đợi 1 xác nhận từ mỗi trạm trở sau những thông báo được gửi đi. Như vậy, mỗi slave xác nhận 1 lần tương ứng những thông báo.
- Thời gian trễ lớn nhất, trước khi slave xác nhận mất đường truyền từ master bằng ít nhất 2 chu kỳ của từng thông báo từ master. (thời gian này sẽ được cài đặt bởi bộ WatchDog Timer của Slave).
- Thời gian trễ lớn nhất, trước khi master xác nhận mất đường truyền từ các slave bằng một chu kỳ truyền thông báo.



- Thời gian trễ lớn nhất, trước khi master xác nhận mất đường truyền từ 1 slave cụ thể bằng những chu kỳ truyền thông báo.

Để truyền thông báo từ master sử dụng 1 ngắt Timer để thay đổi trạng thái của 1 cổng đầu ra của Master.

Nhận tín hiệu đồng bộ sử dụng hàm ngắt ngoài của trạm tớ.

3. Phương pháp đồng bộ thời gian sử dụng UART.

Đặc điểm của phương pháp này sử dụng kênh truyền thông UART để đồng bộ thời gian và dữ liệu, trong đó đồng bộ dữ liệu là một trong những ưu thế của phương pháp này.

Cũng tương tự như phương pháp trên ở trạm chủ sử dụng 1 bộ tạo xung nhịp đồng bộ để gửi tín hiệu cho trạm trở theo 1 chu kỳ nhất định.

Ở trạm trở Slavde được điều hành hoạt động thông qua lệnh đồng bộ nhờ ngắt truyền thông  nhận từ trạm chủ.

Cơ chế hoạt động như sau:

- Tại trạm chủ, xây dựng 1 ngắt Timer truyền tín hiệu đồng bộ và dữ liệu:

Void  MASTER – Update Timer (Void)  interrupt function

.....

MASTER – Send – Tick – Message ( );

.....

- Tại trạm chủ có 1 hàm ngắt truyền thông, sau khi thực hiện xong sẽ giữ 1 thông báo xác nhận về trạm chủ.

Void  SLAVE – Update (void)  interrupt function.

.....

SLAVE – Send – Tick – Message ( );

.....

Cấu trúc của thông báo:

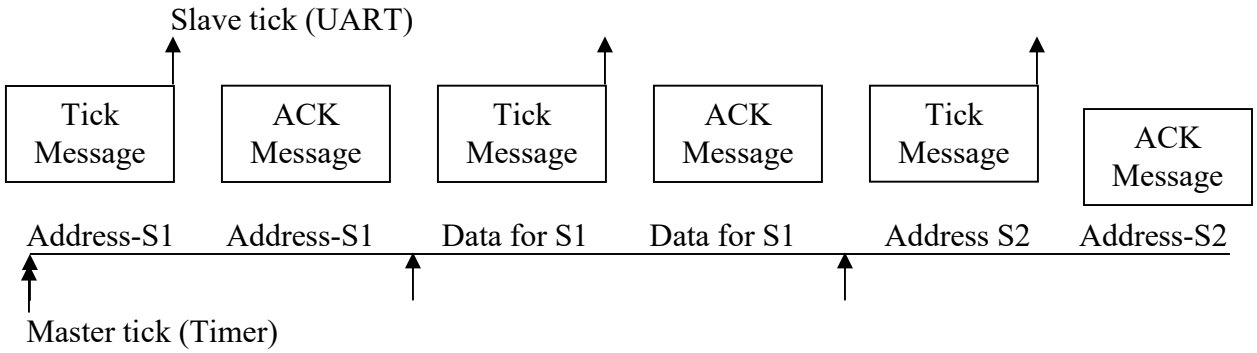
- Mỗi một Slave có 1 ID riêng có mã từ 0x00 đến 0xff

-

- Mỗi một nhịp truyền thông báo từ Master có 2 byte, byte đầu tiên là byte chịu chủ "Address byte" chứa ID của Slave vừa truyền byte thứ 2 là byte thông báo chứa dữ liệu của thông báo.
- Tất cả các Slave sẽ sử dụng hàm ngắt để nhận từng byte của nhịp truyền thông.

Chỉ có Slave có mã ID trùng với ID chứa trong byte trạm chủ. Các nhịp truyền thông sẽ phản hồi lại Master. Dữ liệu phản hồi nhất là sự xác nhận thông báo của Slave đó.

- Mỗi thông báo xác nhận từ Slave cùng có 2 byte. Byte đầu là byte trạm chủ chứa ID của Slave, byte thứ 2 là byte thông báo chứa dữ liệu phản hồi về Master.
- Với dữ liệu thông báo lớn hơn 1 byte thì cần sử dụng kỹ thuật truyền thông báo.




Khi thiết kế cần lưu ý quan hệ giữa tốc độ truyền với dung lượng dữ liệu (Data message) và tổng số nút truyền.