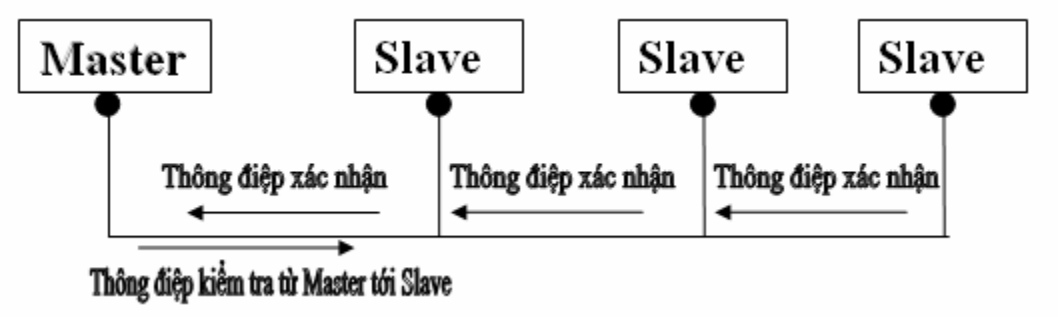
Chương3 :Chương trình phần mềm:

1. Giải pháp xử lý phần mềm:

Giải pháp xử lý phần mềm là xử lý từng tác vụ thông qua phương pháp lập lịch , với sơ đồ giao tiếp của Master và Slave như sau:



Ta cần tạo một vòng "siêu lặp"(Supper Loop) làm cơ sở xử lý cho một chương trình nhúng trong C.

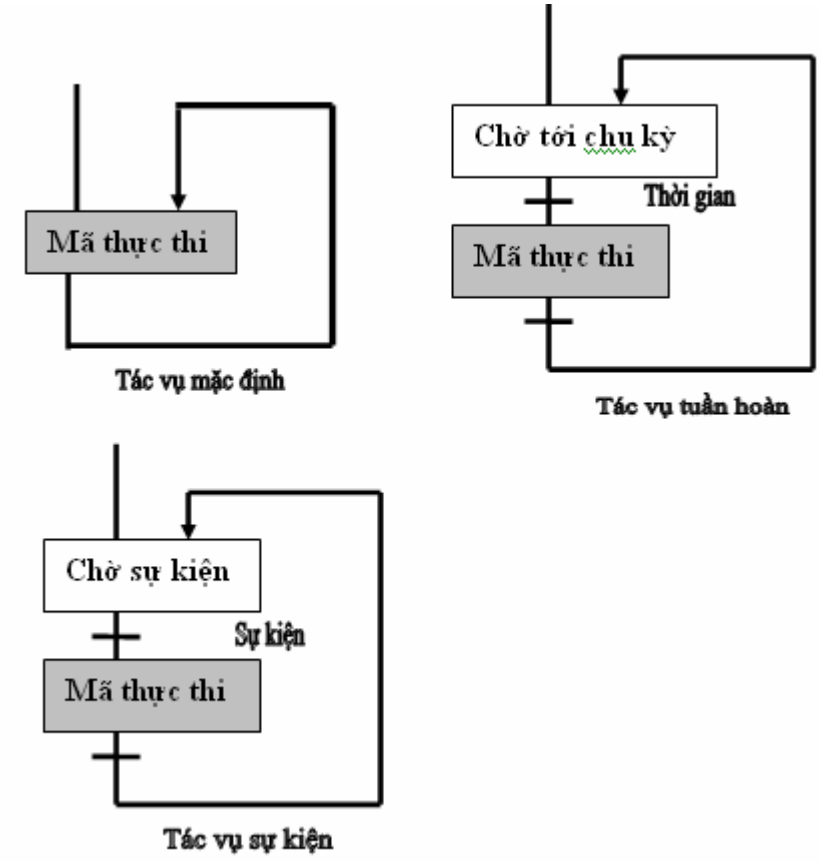
```
void main(void)
{
/* Chuẩn bị cho tTác vụ X */
 X_Init();
 while(1) /* 'không bao giờ kết thúc' (Super Loop) */
 {
   X(); /* Thực hiện  tác vụ */
 }
}
```

Trong hàm **main** trên có từng tác vụ xảy ra mà ta cần phải xử lý chúng.

Khái niệm tác vụ (task) cũng hay được sử dụng bên cạnh quá trình tính toán. Có thể nói, tác vụ là một nhiệm vụ xử lý thông tin trong hệ thống, có thể thực hiện theo cơ chế tuần hoàn (periodic task) hoặc theo sự kiện (event task). Các dạng tác vụ qui định trong chuẩn IEC 61131-3 (Programmable Controllers – Part3: Programming Languages) được minh họa trên hình sau.

Ví dụ : Một tác vụ thực hiện nhiệm vụ điều khiển cho một hoặc nhiều mạch vòng kín có chu kỳ trích mẫu giống nhau. Hoặc, một

tác vụ có thể thực hiện nhiệm vụ điều khiển logic, điều khiển trình tự theo các sự kiện xảy ra. Tác vụ có thể thực hiện dưới dạng một quá trình tính toán duy nhất, hoặc một dãy các quá trình tính toán khác nhau.



Để tổ chức việc thực hiện các tác vụ được hiệu quả, một hệ thống thời gian thực ta cần các phương pháp lập lịch. Trước hết, cơ chế lập lịch thực hiện cho các tác vụ có thể được thực hiện theo hai cách:

* Lập lịch tĩnh: thứ tự thực hiện các tác vụ không thay đổi mà được xác định trước khi hệ thống đi vào hoạt động.

* Lập lịch động: Hàm xác định lịch sau khi hệ thống đã đi vào hoạt động.

Sau khi xác định được cơ chế lập lịch, Hàm cần sử dụng một sách lược lập lịnh (strategy) để áp dụng đối với từng tình huống cụ thể. Có thể chọn một trong những cách sau:

* FIFO (First In First Out): một tác vụ đến trước sẽ được thực hiện trước.

* Mức ưu tiên cố định/động: tại cùng một thời điểm, các tác vụ được đặt các mức ưu tiên cố định hoặc có thể thay đổi nếu cần.

* Preemptive: còn gọi là sách lược chen hàng, tức là chọn một tác vụ để thực hiện trước các tác vụ khác.

* Non-preemptive: còn gọi sách lược không chen hàng. Các tiến trình được thực hiện bình thường dựa trên mức ưu tiên của chúng.

Việc tính mức ưu tiên của mỗi tiến trình được thực hiện theo một trong số các thuật toán sau:

* Rate monotonic: tác vụ nào càng diễn ra thường xuyên càng được ưu tiên.

* Deadline monotonic: tác vụ nào càng gấp, có thời hạn cuối càng sớm càng được ưu tiên.

* Least laxity: tác vụ nào có tỷ lệ thời gian tính toán/thời hạn cuối cùng(deadline) càng lớn càng được ưu tiên.

Bên cạnh phương pháp lập lịch, cơ chế xử lý thời còn đặt ra nhiều vấn đề khác nữa như quản lý và đồng bộ hóa việc sử dụng tài nguyên, giao tiếp liên quá trình, ... Mỗi hệ thống điều khiển là một hệ thời gian thực.

Có thể nói, tất các các hệ thống điều khiển là hệ thời gian thực. Ngược lại, một số lớn các hệ thống thời gian thực là các hệ thống điều khiển. Không có hệ thống điều khiển nào có thể hoạt động bình thường nếu như nó không đáp ứng được các yêu cầu về thời gian, bất kể là hệ thống điều khiển nhiệt độ, điều khiển áp suất, điều khiển lưu lượng hay điều khiển chuyển động. Một bộ điều khiển phải đưa ra được tín hiệu điều khiển kịp thời sau một thời gian nhận được tín hiệu đo để đưa quá trình kỹ thuật về trạng thái mong muốn. Một mạng truyền thông trong một hệ thống điều khiển có tính năng thời gian thực phải có khả năng truyền tin một cách tin cậy và kịp thời đối với các yêu cầu của các bộ điều khiển, các thiết bị vào/ra, các thiết bị đo và thiết bị chấp hành. Tính năng thời gian thực của một hệ thống điều khiển phân tán không chỉ phụ thuộc vào tính năng thời gian thực của từng thành phần trong hệ thống, mà còn phụ thuộc vào sự phối hợp hoạt động giữa các thành phần đó.

Trong thực tế, yêu cầu về tính thời gian thực đối với mỗi ứng dụng điều khiển cũng có các đặc thù khác nhau, mức độ ngặt nghèo khác nhau.

Ví dụ:Cấu trúc phương pháp lập lich và các mảng tác vụ được sử dụng trong này là:

void main(void)

```
{
/* Tạo scheduler */
SCH_Init_ T2();
/* Chuẩn bị cho một tác vụ*/
X_Init();
/*Nạp thời gian cho tác vụ */
SCH_Add_Task(X_Update,a, b);
/* Bắt đầu scheduler */
SCH_Start();
while(1)
{
SCH_Dispatch_Tasks();
}
}

void SCH_Update(void) interrupt
INTERRUPT_Timer_2_Overflow
{
/* Danh sách các tác vụ(task) */
...
}
/* Vùng lưu trữ dữ liệu,nếu có thể được để việc xử lý diễn ra
nhanh
Tổng dung lượng bộ nhớ sử dụng cho một tác vụ(task) là 7byte
*/
typedef data struct
{
/* Con trỏ cho tác vụ( task) ('void (void)' function) */
void (code * pTask)(void);
/* Khoảng trễ cho hàm kế tiếp*/
tWord Delay;
/* Khoảng thời gian lặp lại một chu trình kế tiếp */
tWord Repeat;
/* Set lên 1 (bởi scheduler)khi tác vụ được thực thi */
tByte RunMe;
} sTask;
```

Các mảng tác vụ được ứng dụng trong suốt phương pháp lập lịch

```
/* Mảng tác vụ( task) */
```

sTask SCH_tasks_G[SCH_MAX_TASKS];

*Vấn đề nạp các hàm trong phương pháp lập lịch:

Cấu trúc Sch_Add_Task(Task_Name, Initial_Delay, Task_Interval);

Ví dụ:

SCH_Add_Task(Do_X,1000,0);
Task_ID = SCH_Add_Task(Do_X,1000,0);
SCH_Add_Task(Do_X,0,1000);

Hàm **Do_X()** qui tắc thực thi như sau khoảng lập lịch là 1000 .Tác vụ đầu tiên thực thi ở T=300 thì các tác vụ tiếp theo là ở 1300;2300;…thì ta có cấu trúc lệnh sau:

SCH_Add_Task(Do_X,300,1000);

* Hàm **Disphatcher**

SCH_Dispatch_Tasks()

Hàm Dispatcher có nhiệm vụ khi có một tác vụ sẵn sàng thì lệnh **SCH_Dispatch_Tasks()** sẽ cho phép nó chạy. Hàm này được gọi trong vòng lặp của hàm **main.**

```
void main(void)
{
...
while(1)
{
SCH_Dispatch_Tasks();
}
```

*Vấn đề báo lỗi ta có các lệnh sau:

tByte Error_code_G = 0;

Để sao chép các lỗi này ta dùng các lệnh sau:

```
Error_code_G =
ERROR_SCH_TOO_MANY_TASKS;
Error_code_G =
ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
Error_code_G =
ERROR_SCH_WAITING_FOR_START_COMMAN
D_FROM_MASTER;
```

Error_code_G =
ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
Error_code_G = ERROR_SCH_LOST_SLAVE;
Error_code_G = ERROR_SCH_CAN_BUS_ERROR;

Để báo lỗi ta dung hàm sau:

SCH_Report_Status()

*Các vấn đề báo lỗi khi có sự cố .

- Chúng ta biết rằng mỗi con vi xử lý hoạt động ở một nhiệt độ khác nhau,khi có sự thay đổi về nhiệt độ tốc độ xung clock sẽ thay đổi vì vậy cần có một chương trình đồng bộ xung clock.Trong úng dụng nay ta sử dụng chương trình **SCU SCHEDULER (RS-485) for 8051/52.**Trong chương trình này ta sử dụng một bộ **watchdog timer** sẽ báo tràn khi chu kỳ của con Slave có vẻ chậm(nhanh)hơn so với các khoảng thời gian ta đã định nghĩa.
- Khi truyền dữ liệu giữa Master và Slave mà Slave không nhận được các thong điệp kiểm tra từ Master thì timer sẽ tràn.

Khi truyền dữ liệu giữa Master và Slave thì timer bị tràn và cơ chế lập lịch sẽ gọi hàm "update".Và khi đó một byte dữ liệu sẽ được chuyển đến cho tất cả các Slave (thông qua UART).

```
void MASTER_Update_T2(void) interrupt
INTERRUPT_Timer_2_Overflow
{
...
MASTER_Send_Tick_Message(...);
...
}
```

Khi đã nhận dữ liệu các Slave được ngắt đồng thời lúc này việc lập lịch gọi hàm "update" là ở trong các Slave.Vì vậy các Slave sẽ gửi lại các thông điệp xác nhận cho Master(thông qua UART).

```
void SLAVE_Update(void) interrupt
INTERRUPT_UART_Rx_Tx
{
```

```
...
SLAVE_Send_Ack_Message_To_Master();
...
}
```

Cấu trúc của một chuỗi thông điệp:

- ✓ Mỗi Slave được định nghĩa bằng một địa chỉ ID(0x01 đến 0xFF)
- ✓ Mỗi thông điệp kiểm tra được gửi từ Master đến Slave có độ dài 2 byte được gửi theo từng khoảng (xung) riêng biệt.Byte đầu tiên chứa địa chỉ ID Slave , byte thứ hai chứa dữ liệu của thông điệp.
- ✓ Tất cả các Slave sẽ xãy ra ngắt khi nhận các thông điệp từ Master.
- ✓ Khi các Slave nhận các thông điệp với đúng địa chỉ ID của mình thì khi đó các Slave sẽ trả lời lại bằng các thông điệp xác nhận.
- ✓ Các thông điệp xác nhận cũng có 2 byte và được chuyển đi theo từng khoảng (xung) riêng biệt.Byte dầu chứa địa chỉ ID của Slave đó ,byte thứ hai chứa thông điệp trả lời.
- ✓ Ngoài việc chuyển một byte dữ liệu ta còn có thể chuyển nhiều thông điệp.

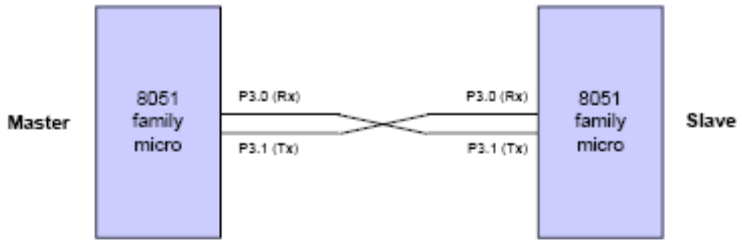
Việc truyền dữ liệu sử dụng họ 8051 cho phép truyền 9-bit được mô tả như sau

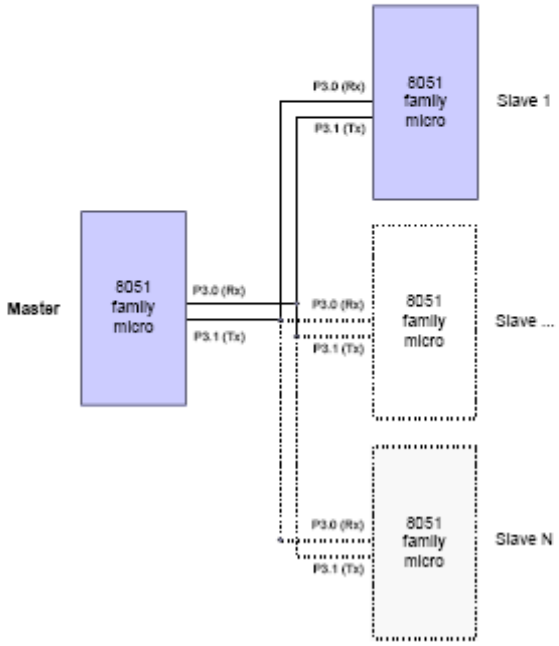
Description	Size (bits)
Data	9 bits
Start bit	1 bit
Stop bit	1 bit
TOTAL	**11 bits / message**

- ✓ Sử dụng UART ở mode 3 truyền/nhận 11-bits.Bit thứ 9 được truyền thông qua bit TB8 của thanh ghi SCON và việc nhận thông qua bit RB8.
- ✓ Byte địa chỉ được nhận dạng bằng cách set bit chung (TB8) lên 1, byte dữ liệu thì set bit đó về 0.

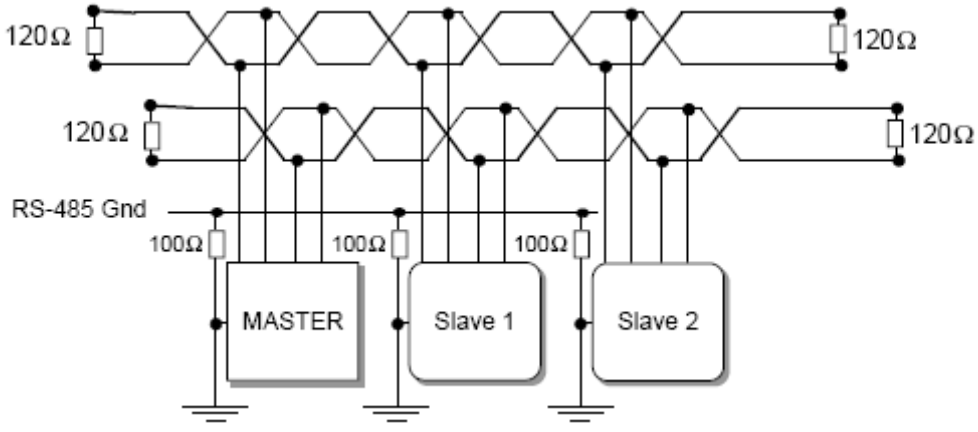
- ✓ Việc truyền dữ liệu có một khoảng trễ giữa timer trên Master và UART là cơ sở cho ngắt trên Slave.

Vấn đề kết nối song song hoặc nối tiếp





Giao tiếp thông qua RS485



2. <u>Phần code hoàn chỉnh:</u>

```
        #include "Main.h"
        #include "Port.h"
        #include "SCU_Bm.H"

        #define PRELOAD01  (65536 - (tWord)(OSC_FREQ /
(OSC_PER_INST * 1063)))
        #define PRELOAD01H (PRELOAD01 / 256)
        #define PRELOAD01L (PRELOAD01 % 256)

        void main(void)
  {
  SCU_B_MASTER_Init_T1_T2(9600);
  SCU_B_MASTER_Start();
  while(1)
    {
    SCH_Dispatch_Tasks();
    }
  }

        void Hardware_Delay_T0(const tWord N)
  {
  tWord ms;
  TMOD &= 0xF0;
  TMOD |= 0x01;
  ET0 = 0;
  for (ms = 0; ms < N; ms++)
    {
    TH0 = PRELOAD01H;
    TL0 = PRELOAD01L;
    TF0 = 0;
     TR0 = 1;
    while (TF0 = = 0);
    TR0 = 0;
    }
  }
```

```
	sbit RS485_Tx_Enable = P3^2;
	sbit RS485_Rx_NOT_Enable = P3^3;
	sbit WATCHDOG_pin = P1^7;
	sbit Network_error_pin = P2^7;
	tByte Tick_message_data_G[NUMBER_OF_SLAVES] =
{'1','2'};
	tByte Ack_message_data_G[NUMBER_OF_SLAVES];
	extern sTask SCH_tasks_G[SCH_MAX_TASKS];
	extern tByte Error_code_G;
	static tByte Current_slave_index_G = 0;
	static bit First_ack_G = 1;
	static bit WATCHDOG_state_G = 0;
	static void SCU_B_MASTER_Reset_the_Network(void);
	static void
SCU_B_MASTER_Shut_Down_the_Network(void);
	static void SCU_B_MASTER_Switch_To_Backup_Slave(const
tByte);
	static void SCU_B_MASTER_Send_Tick_Message(const
tByte);
	static bit  SCU_B_MASTER_Process_Ack(const tByte);
	static void  SCU_B_MASTER_Watchdog_Init(void);
	static void  SCU_B_MASTER_Watchdog_Refresh(void)
reentrant;
	static const tByte
MAIN_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x31,0x32};
	static const tByte
BACKUP_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x31,0x32};
	#define NO_NETWORK_ERROR (1)
	#define NETWORK_ERROR (0)
	#define SLAVE_RESET_INTERVAL 0U
	static tWord
Slave_reset_attempts_G[NUMBER_OF_SLAVES];
	static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] =
{0};
	static bit Message_byte_G = 1;
```

```
        void SCU_B_MASTER_Init_T1_T2(const tWord
BAUD_RATE)
          {
          tByte Task_index;
          tByte Slave_index;
          EA = 0;
          SCU_B_MASTER_Watchdog_Init();
          Network_error_pin = NO_NETWORK_ERROR;
          RS485_Rx_NOT_Enable = 0;
          RS485_Tx_Enable = 1;
          for (Task_index = 0; Task_index < SCH_MAX_TASKS;
Task_index++)
            {
            SCH_Delete_Task(Task_index);
            }
          Error_code_G = 0;
          for (Slave_index = 0; Slave_index <
NUMBER_OF_SLAVES; Slave_index++)
            {
            Slave_reset_attempts_G[Slave_index] = 0;
            Current_Slave_IDs_G[Slave_index] =
MAIN_SLAVE_IDs[Slave_index];
            }
          PCON &= 0x7F;
          SCON = 0xD2;
          TMOD = 0x20;
          TH1 = (256 - (tByte)((((tLong)OSC_FREQ / 100) * 3125)
                / ((tLong) BAUD_RATE * OSC_PER_INST * 1000)));
          TL1 = TH1;
          TR1 = 1;
          TI = 1;
          ES = 0;
          T2CON = 0x04;
          T2MOD = 0x00;
          TH2    = 0xEC;
          RCAP2H = 0xEC;
          TL2    = 0x78;
          RCAP2L = 0x78;
          ET2   = 1;
```

```
        TR2   = 1;
          }

        void SCU_B_MASTER_Start(void)
        {
        tByte Slave_ID;
        tByte Num_active_slaves;
        tByte Id, i;
        bit First_byte = 0;
        bit Slave_replied_correctly;
        tByte Slave_index;
        SCU_B_MASTER_Watchdog_Refresh();
        SCU_B_MASTER_Enter_Safe_State();
        Network_error_pin = NETWORK_ERROR;
        Error_code_G =
ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
        SCH_Report_Status();
        for (i = 0; i < 100; i++)
          {
          Hardware_Delay_T0(30);
          SCU_B_MASTER_Watchdog_Refresh();
          }
        Num_active_slaves = 0;
        Slave_index = 0;
        do {
          SCU_B_MASTER_Watchdog_Refresh();
          First_byte = 0;
          Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];
          TI = 0;
          TB8 = 1;
          SBUF = Slave_ID;
          Hardware_Delay_T0(5);
          if (RI = = 1)
            {
            Id = (tByte) SBUF;
            RI = 0;
            if ((Id = = Slave_ID) && (RB8 = = 1))
              {
              First_byte = 1;
              }
```

```
            }
          TI = 0;
          TB8 = 1;
          SBUF = Slave_ID;
          Hardware_Delay_T0(5);
          Slave_replied_correctly = 0;
          if (RI != 0)
            {
            Id = (tByte) SBUF;
            RI =0;
            if ((Id = = Slave_ID) && (RB8 = = 1) && (First_byte = =
1))
              {
              Slave_replied_correctly = 1;
              }
            }
          if (Slave_replied_correctly)
            {
            Num_active_slaves++;
            Slave_index++;
            }
          else
            {
            if (Current_Slave_IDs_G[Slave_index] !=
BACKUP_SLAVE_IDs[Slave_index])
              {
              Current_Slave_IDs_G[Slave_index] =
BACKUP_SLAVE_IDs[Slave_index];
              }
            else
              {
              Slave_index++;
              }
            }
          } while (Slave_index < NUMBER_OF_SLAVES);
            if (Num_active_slaves < NUMBER_OF_SLAVES)
          {
          Error_code_G =
        ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_STA
        RT;
```

```
    SCU_B_MASTER_Shut_Down_the_Network();
    }
  else
    {
    Error_code_G = 0;
    Network_error_pin = NO_NETWORK_ERROR;
    }
  Message_byte_G = 1;
  First_ack_G = 1;
  Current_slave_index_G = 0;
  EA = 1;
  }

void SCU_B_MASTER_Update_T2(void) interrupt
INTERRUPT_Timer_2_Overflow
  {
  tByte Task_index;
  tByte Previous_slave_index;
  TF2 = 0;
  SCU_B_MASTER_Watchdog_Refresh();
  Network_error_pin = NO_NETWORK_ERROR;
  Previous_slave_index = Current_slave_index_G;
  if (Message_byte_G = = 0)
    {
    Message_byte_G = 1;
    }
  else
    {
    Message_byte_G = 0;
   if (++Current_slave_index_G >= NUMBER_OF_SLAVES)
      {
      Current_slave_index_G = 0;
      }
    }
  if (SCU_B_MASTER_Process_Ack(Previous_slave_index) =
= RETURN_ERROR)
    {
    Network_error_pin = NETWORK_ERROR;
    Error_code_G = ERROR_SCH_LOST_SLAVE;
```

```
          if (++Slave_reset_attempts_G[Previous_slave_index] >=
SLAVE_RESET_INTERVAL)
              {
              SCU_B_MASTER_Reset_the_Network();
              }
            }
          else
            {
            Slave_reset_attempts_G[Previous_slave_index] = 0;
            }


SCU_B_MASTER_Send_Tick_Message(Current_slave_index_G);
          for (Task_index = 0; Task_index < SCH_MAX_TASKS;
Task_index++)
            {
            if (SCH_tasks_G[Task_index].pTask)
              {
              if (SCH_tasks_G[Task_index].Delay = = 0)
                {
                SCH_tasks_G[Task_index].RunMe += 1;

                if (SCH_tasks_G[Task_index].Period)
                  {
                  SCH_tasks_G[Task_index].Delay =
SCH_tasks_G[Task_index].Period;
                  }
                }
              else
                {
                SCH_tasks_G[Task_index].Delay = 1;
                }
              }
            }
          }

        void SCU_B_MASTER_Send_Tick_Message(const tByte
SLAVE_INDEX)
          {
          tLong Timeout;
```

```
        tByte Slave_ID = (tByte)
Current_Slave_IDs_G[SLAVE_INDEX];
        if (Message_byte_G == 0)
          {
          Timeout = 0;
          while ((++Timeout) && (TI = = 0));
          if (Timeout = = 0)
            {
            Error_code_G = ERROR_USART_TI;
            return;
            }
          TI = 0;
          TB8 = 1;
          SBUF = Slave_ID;
          }
        else
          {
          Timeout = 0;
          while ((++Timeout) && (TI = = 0));
          if (Timeout = = 0)
            {
            Error_code_G = ERROR_USART_TI;
            return;
            }
          TI = 0;
          TB8 = 0;
          SBUF = Tick_message_data_G[SLAVE_INDEX];
          }
        }

      bit SCU_B_MASTER_Process_Ack(const tByte Slave_index)
        {
        tByte Message_contents;
        tByte Slave_ID;
        if (First_ack_G)
          {
          First_ack_G = 0;
          return RETURN_NORMAL;
          }
        Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];
```

```
  if (RI = = 0)
    {
    Network_error_pin = NETWORK_ERROR;
    return RETURN_ERROR;
    }
  Message_contents = (tByte) SBUF;
  RI = 0;
  if (Message_byte_G = = 1)
    {
    if (RB8 = = 1)
    {
       if (Slave_ID = = (tByte) Message_contents)
          {
          return RETURN_NORMAL;
          }
       }
     Network_error_pin = NETWORK_ERROR;
     return RETURN_ERROR;
     }
   else
     {
     Ack_message_data_G[Slave_index] = Message_contents;
      return RETURN_NORMAL;
     }
  }

void SCU_B_MASTER_Reset_the_Network(void)
  {
  EA = 0;
  while(1);
  }

void SCU_B_MASTER_Shut_Down_the_Network(void)
  {
  EA = 0;
  Network_error_pin = NETWORK_ERROR;
  SCH_Report_Status();
  while(1)
    {
    SCU_B_MASTER_Watchdog_Refresh();
```

```
    }
  }

void SCU_B_MASTER_Watchdog_Init(void)
  {
  }

void SCU_B_MASTER_Watchdog_Refresh(void) reentrant
  {
  if (WATCHDOG_state_G = = 1)

  {
    WATCHDOG_state_G = 0;
    WATCHDOG_pin = 0;
    }
  else
    {
    WATCHDOG_state_G = 1;
    WATCHDOG_pin = 1;
    }
  }
```

VI. Tổng kết:

1. Tổng kết về ý tưởng và khả năng ứng dụng thực tiễn của đề tài:

Qua việc sử dụng kết hợp 2 chuẩn truyền thông giao tiếp thông dụng hiện nay và sử dụng phương thức thu thập, phân tán dữ liệu master-slave, ta thấy rằng việc thu thập và phân tán thông tin được thực hiện nhanh chóng, dễ dàng hơn và đặc biệt là gần như cùng 1 lúc ta có thể giao tiếp với nhiều module trong mối quan hệ chặt chẽ, có thứ tự.Vi điều khiển master có thể giao tiếp với khoảng 256 module trên đường dây truyền thông RS-485 với thời gian thu phát dữ liệu giữa 2 module kế tiếp nhau là không đáng kể.

Vì vậy ý tưởng của đề tài này có thể ứng dụng vào mạng truyền thông trong các nhà máy, xí nghiệp gồm nhiều bộ phận làm việc ở cách xa nhau mà việc thu thập và phân tán thông tin theo phương thức thủ công là không khả thi. Các bộ phận có trách nhiệm thu thập và phân tán dữ liệu chỉ cần ở tại vị trí của master để cập nhật và kiểm soát thông tin. Họ chỉ cần thiết kế hệ thống hợp lý phù hợp với nhu cầu và không gian của địa điểm làm việc là có thể giao tiếp với các bộ phận khác dễ dàng và nhanh chóng mà lại có được nội dung thông tin chính xác.

2. Phương hướng phát triển của đề tài:

Ngày nay sự phát triển vũ bão của hệ thống truyền thông Internet đã làm bùng nổ một phương thức truyền thông mới nhanh hơn gấp nhiều lần, thông tin chính xác và rất thuận tiện cho người làm việc. Cho nên phương hướng phát triển của đề tài là thông tin từ vi điều khiển được cập nhật và truyền qua máy vi tính sau đó sử dụng hệ thống mạng Internet thông qua địa chỉ IP để truyền thông tin đến một người có trách nhiệm cao nhất ở bất kì 1 nơi nào đó trên thế giới xử lý và truyền thông tin ngược lại. Việc này vừa làm giảm thiểu được nguồn nhân công cố định tại nơi làm việc vừa làm giảm được không gian nhà máy.

Tóm lại phương thức này sẽ trở nên phổ biến rộng rãi trong thời gian không xa tới đây.