

CHƯƠNG 5: LẬP TRÌNH C AVR

5.1 Giới thiệu chung

Trong chương 4 ta đã khảo sát lập trình bằng hợp ngữ cho AVR. Hợp ngữ là ngôn ngữ cấp thấp sát với ngôn ngữ máy nhất. Khi lập trình bằng hợp ngữ người lập trình có thể định lượng dung lượng chương trình chiếm trong bộ nhớ Flash, tính toán chính xác thời gian thực các tác vụ của MCU, điều khiển trực tiếp đến thiết kế phần cứng... Tuy nhiên lập trình hợp ngữ sẽ bị hạn chế trong các tác vụ tính toán, truy xuất mảng bộ nhớ, và nhát là diễn đạt ý nghĩa các lệnh rõ ràng dễ hiểu!

Ngôn ngữ C là ngôn ngữ cấp cao hơn hợp ngữ, có ưu điểm hơn hợp ngữ ở các vấn đề sau:

- Trình bày đơn giản, ngắn gọn, dễ hiểu
- Thực hiện các tác vụ tính toán, xử lý mảng dữ liệu chỉ bằng các lệnh đơn giản
- Không cần hiểu rõ tập lệnh, định vị địa chỉ của MCU, chỉ cần nắm các từ khóa, ký hiệu quy định theo từng họ MCU (do trình biên dịch C quy định)

Tuy nhiên nhược điểm của ngôn ngữ C (chính là các ưu điểm của hợp ngữ) ở các vấn đề sau:

- Trình biên dịch C cho ra file mã máy dung lượng khá cao so với hợp ngữ, không thích hợp với các họ MCU có dung lượng bộ nhớ Flash thấp!
- Không tính toán được thời gian thực 1 tác vụ của MCU
- Không nắm được tác động điều khiển trực tiếp đến phần cứng!

Các trình biên dịch C hiện nay đều có công cụ biên dịch chéo từ ngôn ngữ C sang hợp ngữ, từ hợp ngữ sang mã máy, hon nura trong môi trường lập trình C cho phép lập trình hợp ngữ. Do đó nếu hiểu rõ cả 2 ngôn ngữ hợp ngữ và C, ta sẽ chủ động hoàn toàn trong thiết kế phần cứng và phần mềm, tận dụng được ưu thế của cả 2 ngôn ngữ lập trình!

Trong chương này sẽ trình bày lập trình bằng ngôn ngữ C cho họ AVR theo trình biên dịch GCC-AVR của ATmel, nay là của hãng Microchip. Các từ khóa, ký hiệu đều theo quy định của GCC-AVR. Việc soạn thảo chương trình C có thể thực hiện trên bất kỳ phần mềm soạn thảo văn bản nào phù hợp thông dụng như Word, Notepad trong Windows... Trình biên dịch, mô phỏng ATmel Studio 7 bao gồm biên soạn, biên dịch hợp ngữ C, mô phỏng, nạp code cho chip AVR được sử dụng cho các ví dụ, bài tập trong giáo trình này và trong phần Thí nghiệm Vi xử lý.

❖ Giới thiệu định dạng một chương trình C

Sau đây ta xem một ví dụ tiêu biểu định dạng một chương trình C.

Ví dụ 5.1:

```
#include    <avr/io.h> [1]
#define P_out  PC0          [2]
/* Hàm delay tạo trễ Td(μs)=6xn MC */ [3]
void delay_ms(unsigned int n)// hàm delay ms [4]
{
    unsigned int i; [5]
    for(i=0;i<=n;i++); // Td(μs)=6xn MC [6]
}
int main() [7]
{
    DDRC=0xff ;//PC0 output [8]
    PORTC=0xfe ;//PC0=0 [9]
    while(1)   //lặp vòng vô hạn [10]
    {
        PORTC=PORTC^(1<<P_out); //đảo bit P_out [11]
        delay_ms(13333); //delay 10ms [12]
    }
}
```

- Dòng [1] khai báo từ khóa tiền xử lý (preprocessor) #include báo trình biên dịch bao gồm file tiêu đề (header file) avr/io.h định nghĩa các ký hiệu IORs của AVR có sẵn trong thư mục thư viện
- Dòng [2] dùng tiền xử lý #define định nghĩa ký hiệu P_out=PC0, bit0 PortC
- Dòng [3] chú giải ý nghĩa đoạn chương trình sau đó

- Dòng [4] khai báo hàm tạo thời gian trễ tên delay_ms theo biến nguyên không dấu ký hiệu unsigned int n
- Dòng [5] khai báo biến nguyên không dấu unsigned int i sử dụng trong phạm vi hàm delay_ms
- Dòng [6] là lệnh lập vòng tạo trễ trong hàm delay_ms
- Dòng [7] bắt đầu chương trình(hàm) chính thường ký hiệu int main() hoặc void main()
- Các dòng từ [8] đến [12] biểu diễn các phát biểu hay biểu thức trong chương trình chính

❖ Các ký hiệu đặc biệt sử dụng trong chương trình C:

- Dấu (;) : báo trình biên dịch kết thúc một phát biểu hay biểu thức
- Dấu ({...}) : nội dung các phát biểu hay biểu thức nằm trong dấu ({ }) là giới hạn các phát biểu của một hàm, hoặc được xem như 1 phát biểu.
- Dấu (//) chỉ báo dòng chú giải ở phía sau hoặc dấu /*... */ báo đoạn chú giải nằm bên trong, trình biên dịch sẽ bỏ qua khi biên dịch
- Khoảng trắng : có thể là dấu cách(space bar), Tab, dòng trống phân biệt các thành phần trong một phát biểu, sử dụng trong phần chú giải

❖ Danh hiệu(Identifiers) và các từ khóa(Key words):

- Danh hiệu là tên đặt cho hàm, biến, hằng số bắt đầu bằng chữ cái hoặc dấu (_), theo sau là chữ cái, chữ số hoặc dấu ()
- Nói chung không giới hạn số ký tự cho 1 danh hiệu, tuy nhiên một số phần mềm giới hạn 32 ký tự
- Danh hiệu nhạy kiếu chữ in hoa và thường
- Một số từ khóa đặc biệt dành riêng cho trình biên dịch biểu diễn các định nghĩa, chức năng...
- Không được sử dụng các từ khóa làm danh hiệu (trình biên dịch sẽ báo lỗi khi biên dịch).
- Một số từ khóa tiêu biểu như liệt kê trong Bảng 5.1.

Bảng 5.1: Các từ khóa tiêu biểu

auto	defined	float	long	static	while
break	do	for	register	struct	
bit	double	funcused	return	switch	
case	eeprom	goto	short	typedef	
char	else	if	signed	union	
const	enum	inline	sizeof	unsigned	
continue	extern	int	sfrb	void	
default	flash	interrupt	sfrw	volatile	

5.2 Biến(Variabiles) và Hằng(Constants)

Các phát biểu hay biểu thức luôn có các thành phần là biến và hằng. Mỗi loại biến và hằng đều được định nghĩa kiểu dữ liệu(data types) xác định tầm giá trị tương ứng với số bit biểu diễn.

5.2.1 Kiểu dữ liệu

Có 4 kiểu dữ liệu biểu diễn trong ngôn ngữ C:

1. Kiểu cơ bản(Basis types): là các kiểu biểu diễn số học gồm kiểu số nguyên(integer types)) và kiểu dấu chấm động(floating point types)
2. Kiểu liệt kê(Enumerated types): cũng là kiểu số học, định nghĩa biến gán cho 1 số xác định sử dụng trong suốt chương trình
3. Kiểu không có giá trị(void types): không có giá trị xác định
4. Kiểu dẫn xuất(derived types): bao gồm kiểu con trỏ(pointer types), kiểu mảng(array types), kiểu cấu trúc(structure types), kiểu hợp nhất(union types), kiểu hàm(function types)

Kiểu mảng, cấu trúc, hợp nhất tổng hợp nhiều kiểu dữ liệu, còn kiểu hàm liên quan đến kiểu dữ liệu của giá trị trả về sau khi thực hiện hàm. Sau đây ta sẽ tìm hiểu kiểu cơ bản và kiểu không có giá trị, còn các kiểu còn lại sẽ được trình bày chi tiết trong các mục sau.

5.2.1.1 Kiểu cơ bản

Bảng 5.2 mô tả định dạng các kiểu số nguyên và dấu chấm động, tầm giá trị và số bit biểu diễn trong C-AVR:

Bảng 5.2: Mô tả các kiểu số nguyên, dấu chấm động

Kiểu	Độ dài (byte)	Tầm giá trị
char	1	-128÷127 hoặc 0÷255
unsigned char	1	0÷255
signed char	1	-128÷127
int	2	-32768÷32767 hoặc 0÷65535
unsigned int	2	0÷65535
signed int	2	-32768÷32767
long	4	-2,147,483,648÷2,147,483,647 hoặc 0÷4,294,967,295
unsigned long	4	0÷4,294,967,295
signed long	4	-2,147,483,648÷2,147,483,647
float	4	$\pm 3.402e-38 \div \pm 3.402e38$
double	4	$\pm 3.402e-38 \div \pm 3.402e38$

Ví dụ 5.2: Xem khai báo các biến sau và tìm giá trị trả về của các biến theo hệ Hex:

```
#include <avr/io.h>
char a;
signed char b;
int c;
short int d;
signed long e;
float f;
double g;
int main(void)
{
    a=127;
    b=-127;
    c=32767;
    d=-32767;
    e=-2147483647;
    f=-3.4e38;
    g=-3.4e-38
}
```

Giải:

```
a=0x7F
b=0x81
c=0x7FFF
d=0x8001
e=0x800001
f=0x7F7FC99E
g=0x81391D15
```

5.2.1.2 Kiểu không có giá trị

Kiểu không có giá trị khai báo bằng từ khóa **void**

1. Hàm trả về không có giá trị - Có một số hàm sau khi thực thi xong không cần trả về giá trị, ví dụ như hàm tạo trễ trong ví dụ 5.1 khai báo **void delay_ms(unsigned int n)**
2. Hàm không sử dụng các thông số/biến từ bên ngoài, còn gọi là thông số hình thức(argument), ví dụ **int main(void)** hay **int main()**

3. Con trả không có giá trị **void *** trả về địa chỉ của đối tượng,không phải kiểu của nó,ví dụ hàm phân bổ vùng nhớ **void *malloc(size_t size)** trả về con trả không có giá trị có thể gán cho bất kỳ kiểu dữ liệu nào.

5.2.2 Biến(variables)

Biến là một danh hiệu(tên)được gán cho một địa chỉ ô nhớ cụ thể trong vùng nhớ xác định Flash, EEPROM hay SRAM đã khai báo đính kèm.Nếu không có khai báo vùng nhớ,trình biên dịch mặc định là vùng SRAM và địa chỉ đầu là 0x100 cho MCU324P.Biến là thành phần của phát biểu hay biểu thức,do đó phải khai báo kiểu dữ liệu và vùng nhớ lưu trữ như đã trình bày ở trên.Trình biên dịch sẽ trả về đúng kiểu dữ liệu đã khai báo của biến trước đó.Các kiểu dữ liệu biểu diễn bằng số của biến thông dụng như mô tả trong Bảng 5.2.Ngoài ra biến còn được khai báo bằng các kiểu dữ liệu khác đã mô tả ở mục trước,sẽ được khảo sát sau.

Ví dụ 5.3: Đoạn chương trình sau khai báo các biến và tính toán các biểu thức theo các biến:

```
#include <avr/io.h>
char a,b;          //a,b biến kiểu char 1 byte
int c;             //c biến kiểu int 2 byte
float d,e,f;       //d,e,f biến kiểu float 4 byte
int main(void)
{
    while (1)
    {
        a=57;    //a=57=0x39
        b=93;    //b=93=0x5d
        d=1.732;//d=1.732=3fddb22d
        e=1.245;//e=1.245=3f9f5c29
        c=a*b;   //c=57x93=5301=0x14b5
        f=d/e;   //f=1.732/1.245=1.391164658634538...=0x3fb211af
    }
}
```

Biến được phân thành 2 loại: biến cục bộ(local variables) và biến toàn cục(global variables)

5.2.2.1 Biến cục bộ

Biến cục bộ được khai báo trong phạm vi định nghĩa hàm và chỉ sử dụng trong phạm vi hàm đã khai báo.Do đó,tu có thể khai báo biến trùng tên gọi trong các hàm khác nhau,trình biên dịch vẫn phân biệt sử dụng biến trong hàm đã khai báo độc lập nhau.

➤ Trình biên dịch C-AVR thường mặc định sử dụng các GPRs làm các biến cục bộ.

5.2.2.2 Biến toàn cục

Biến toàn cục được khai báo ngoài chương trình chính(thông thường đặt trước dòng **int main()**), và được sử dụng trong toàn chương trình chính và các hàm có trong chương trình.Biến toàn cục được định nghĩa 1 lần duy nhất và không được phép định nghĩa lại.Trường hợp tên biến toàn cục trùng với biến cục bộ trong hàm,trình biên dịch sẽ sử dụng biến cục bộ và loại bỏ biến toàn cục trong hàm.

➤ Trình biên dịch C-AVR thường mặc định sử dụng các ô nhớ SRAM làm các biến toàn cục,bắt đầu từ địa chỉ 0x100 với MCU324P.

Chi tiết về biến cục bộ và biến toàn cục sẽ được trình bày trong mục Hàm ở phần sau.

Ví dụ 5.4: Xem đoạn chương trình sau có khai báo biến cục bộ và toàn cục:

```
#include <avr/io.h>
unsigned char a;      //a kiểu unsigned char là biến toàn cục
unsigned int b,c;     //b,c kiểu unsigned int là biến toàn cục
void mul_a()          //hàm nhân a
{
    unsigned char i; //i kiểu unsigned char là biến cục bộ
    i=10 ;           //đặt i=10
    b=a*i ;         //b chứa kết quả nhân axi
}
```

```

void add_a()          //hàm cộng a
{
    unsigned int j;  //j kiểu unsigned int là biến cục bộ
    j=200;           //đặt j=200
    c=a+j;           //c chứa kết quả cộng a+j
}
int main()
{
    a=10 ;           //đặt a=10
    mul_a();          //gọi hàm nhân a trả kết quả trong b
    add_a();          //gọi hàm cộng a trả kết quả trong c
    while(1);         //lặp vòng tại chỗ vô hạn
}

```

5.2.3 Hằng(Constants)

Hằng là các giá trị cố định được sử dụng trong chương trình và chương trình không thể thay đổi khi thực thi.Hằng còn được gọi chung là ký tự(literals).

Hằng có thể là bất kỳ kiểu dữ liệu nào như hằng số nguyên,hằng số dấu chấm động,hằng ký tự,hằng chuỗi,hằng liệt kê...Hằng có thể xem như biến được gán giá trị cố định và không được định nghĩa lại.

5.2.3.1 Hằng số nguyên

Hằng số nguyên được biểu diễn với hệ thập phân,nhi phân,bát phân(octal),thập lục phân(hex).Hệ thập phân không có ký hiệu,hệ nhị phân có ký hiệu 0b ở đầu,hệ octal có ký hiệu 0 ở đầu,hệ hex có ký hiệu 0x ở đầu.Có thể thêm hậu tố u hoặc U chỉ số không dấu unsigned,l hoặc L chỉ kiểu dữ liệu long,hoặc kết hợp UL chỉ kiểu dữ liệu unsigned long.Hậu tố không nhạy kiểu chữ in hoa hoặc thường.

Ví dụ 5.5: Xem đoạn chương trình sau và tìm giá trị hằng số hệ hex và số byte gán cho các biến tương ứng.

```

#include <avr/io.h>
unsigned char a,b;
unsigned int c,d;
long e,f;
int main()
{
    while(1)          //lặp vòng vô hạn
    {
        a=156 ;       //hệ thập phân
        b=0123;        //hệ octal
        c=0x1234;      //hệ hex
        d=0b01101101; //hệ nhị phân
        e=1234u;        //hệ thập phân không dấu
        f=1234l;        //hệ thập phân kiểu long
    }
}

```

Giải:

a=0x9c	1 byte
b=0x53	1 byte
c=0x1234	2 byte
d=0x006d	2 byte
e=0x000004d2	4 byte
f=0x000004d2	4 byte

5.2.3.2 Hằng số dấu chấm động

Hằng số dấu chấm động gồm phần nguyên,dấu chấm thập phân,phần thập phân,phần lũy thừa.Có thể biểu diễn dưới dạng số thập phân hoặc dạng lũy thừa mũ 10.

Ví dụ 5.6: Các hằng số dấu chấm động:

```
a=3.14159      //a=0x40490fd0
b=314159e-5
c=314159E-5
```

5.2.3.3 Hằng ký tự(character constants)

Các hằng ký tự được đặt trong dấu ('...')biểu diễn mã ASCII của ký tự, chỉ cần khai báo kiểu char 1 byte. Ví dụ như 'A'=0x41,'a'=0x61...

Ngoài ra trong ngôn ngữ C còn định nghĩa các hằng ký tự đặc biệt gọi là (escape sequence)được liệt kê trong Bảng 5.3.

Bảng 5.3: Các ký tự escape sequence

Escape sequence	Ý nghĩa
\\	Ký tự \
\'	Ký tự '
\"	Ký tự "
\?	Ký tự ?
\a	Báo động/chuông(Alert/Bell)=0x07
\b	Xóa lùi(Backspace)=0x08
\f	Ngắt trang(Form feed)=0x0c
\n	Dòng mới(New line)=0x0a
\r	Xuống dòng(Carriage return)=0x0d
\t	Tab ngang(Horizontal tab)=0x09
\v	Tab dọc(Vertical tab)=0x0b
\ooo	số hệ octal 2 đến 3 chữ số
\xhh...	số hệ hex

Ví dụ 5.7: Biểu diễn các hằng ký tự:

```
a='b' ;a=0x62
b='!' ;b=0x21
c='xfd' ;c=0xfd
d='\r' ;d=0x0d
```

5.2.3.4 Hằng chuỗi(String constants)

Hằng chuỗi gồm 1 chuỗi hằng ký tự được đặt trong dấu ("..."). Các ký tự bao gồm ký tự escape sequence viết liền nhau. Trường hợp chuỗi ký tự dài quá 1 dòng,sử dụng dấu nối dòng (\) báo trình biên dịch chuỗi ký tự liền 1 dòng.

Ví dụ 5.8: Các chuỗi ký tự sau là giống nhau:

“Hello!\rXin chào!\xfd”

hoặc:

“Hello!””\x0d””
“Xin chào!””\xfd”

5.2.3.5 Định nghĩa hằng

Để định nghĩa hằng ta có thể sử dụng tiền xử lý **#define** hoặc từ khóa **const**. Trường hợp sử dụng từ khóa **const** ta phải khai báo kiểu dữ liệu cho hằng.

❖ **Cú pháp:**

```
#define danh hiệu giá trị
const kiểu dữ liệu danh hiệu1=giá trị1[,danh hiệu2=giá trị2,...];
```

Ví dụ 5.9: Các định nghĩa hằng cho các danh hiệu sau:

```
#define length 10
#define CR 0xd
const int Tf=1000,SR_buf=0x100;
const char a=0xfd,b=0b10011100;
```

❖ Câu hỏi ôn tập

1. Khai báo kiểu dữ liệu cho các biến sau,biết rằng tầm giá trị của chúng: a=0-255,b=-128-+127, c=0-65535,d=0,000 -0,999.
2. Các biến sau đây có các giá trị : x=0xfa,y=0b00101101,z=1.23e-1,t=34.Khai báo kiểu dữ liệu phù hợp cho các biến trên.
3. Khai báo các ký hiệu sau: dat=0x3f,S_addr=0x10df,AB=""AB",num=10
4. Khai báo các ký hiệu sau :Import=PIN_A,IO_dir=DDRA,Outport=PORTA
5. Khai báo chuỗi ký tự sau Hello! và kết thúc chuỗi bằng mã Null=0x00
6. Biến toàn cục và biến cục bộ thường được đặt vị trí khai báo ở đâu?Chúng khác nhau chỗ nào?

5.2.4 Kiểu bộ nhớ(Memory types)

Khi khai báo biến và hằng,ngoài việc định nghĩa kiểu dữ liệu còn phải định nghĩa kiểu bộ nhớ.Họ AVR có cấu trúc 3 vùng bộ nhớ là vùng chương trình thuộc Flash ROM ,vùng dữ liệu thuộc SRAM hay EEPROM.Tùy vào mục đích yêu cầu cụ thể,có thể đặt biến và hằng vào 1 trong 3 vùng bộ nhớ trên.

Đối với biến giá trị có thể thay đổi,nên trình biên dịch mặc định đặt biến trong vùng nhớ SRAM.Trường hợp cần lưu giá trị biến khi cắt điện,nên đặt biến trong vùng nhớ EEPROM.

Đối với hằng có thể đặt trong cả 3 vùng nhớ trên.Khi khai báo hằng kiểu dữ liệu chuỗi(strings) hoặc mảng(arrays)mà không định nghĩa kiểu bộ nhớ,trình biên dịch sẽ cắt chuỗi/mảng dữ liệu trong vùng nhớ chương trình,khi khởi động chương trình sẽ chép chuỗi/mảng dữ liệu này vào vùng nhớ SRAM.Như vậy cả bộ nhớ Flash và SRAM đều chứa chuỗi/mảng dữ liệu,sẽ gây lãng phí vùng nhớ!

Trong trường hợp dữ liệu hằng không thay đổi,ta nên cắt trực tiếp trong vùng nhớ Flash để giải phóng vùng nhớ SRAM.Trường hợp dữ liệu hằng ít thay đổi và cần lưu khi cắt điện,ta nên cắt trong vùng nhớ EEPROM.

Trình biên dịch ký hiệu vùng nhớ chương trình là “text”,vùng nhớ SRAM là “data”,vùng nhớ EEPROM là “eprom”.

Khai báo kiểu bộ nhớ cho hằng theo cú pháp sau:

```
const kiểu dữ liệu danh hiệu __attribute__((section(".kiểu bộ nhớ")))=giá trị;
```

- kiểu bộ nhớ=data trỏ vùng nhớ SRAM
- kiểu bộ nhớ=text trỏ vùng nhớ Flash ROM
- kiểu bộ nhớ=eprom trỏ vùng nhớ EEPROM

➤ Mặc định là kiểu bộ nhớ data(SRAM) nếu không khai báo __attribute__(...)

Trường hợp khai báo hằng kiểu bộ nhớ Flash có thể dùng cú pháp sau:

```
#include <avr/pgmspace.h>
```

```
const kiểu dữ liệu danh hiệu PROGMEM=giá trị;
```

Trường hợp khai báo hằng kiểu bộ nhớ EEPROM có thể dùng cú pháp sau:

```
#include <avr/eeprom.h>
```

```
const kiểu dữ liệu danh hiệu EEMEM=giá trị;
```

Gán giá trị hằng khai báo kiểu bộ nhớ flash hay eeprom có thể trực tiếp hoặc gián tiếp dùng con trỏ địa chỉ(pointers),sẽ trình bày ở phần sau.

Ví dụ 5.10: Khai báo dãy hằng xác định kiểu bộ nhớ:

```
#include <avr/pgmspace.h> [1]
```

```
const char x_ep[] __attribute__((section(".eprom")))= {0xc0,0xf9,0xa4,0xb0,0x99,\ [2]
```

```
0x92,0x82,0xf8,0x80,0x90,0x88,0x83,0xc6,0xa1,0x86,0x8e};
```

```
const char str[] PROGMEM="Hello! "; [3]
```

- Dòng [1] báo trình biên dịch bao gồm file tiêu đề để sử dụng từ khóa PROGMEM
- Dòng [2] khai báo dãy dữ liệu x_ep[] gồm 16 giá trị đặt trong bộ nhớ EEPROM,địa chỉ khởi động mặc định là 0x0000.
- Dòng [3] khai báo dãy dữ liệu str[] gồm các ký tự trong dấu (“...”)đặt trong bộ nhớ Flash,trình biên dịch sẽ ẩn định địa chỉ khi biên dịch.
- Kiểu dữ liệu dãy trong ví dụ trên sẽ được trình bày ở phần sau.

5.2.5 Hoạt động các thanh ghi I/O(IORs)

C-AVR có thể hiểu và tương tác trực tiếp đến các IORs và các thanh ghi I/O mở rộng Ex-IORs.Các IORs ,Ex-IORs được xem là các biến với tên biến chính là các ký hiệu IORs do nhà sản xuất quy định.

Người lập trình chỉ cần biết ký hiệu và độ dài(1 hoặc 2 byte)của các thanh ghi và lập trình tương tự như đối với biến và không cần khai báo.Tất nhiên để sử dụng được các ký hiệu IORs,Ex-IORs ta phải có khai báo file tiêu đề đã định nghĩa các ký hiệu IORs,ở đầu chương trình:

#include <avr/io.h>

➤ **Lưu ý: Quy định ký hiệu các IORs đều viết chữ in hoa**

Ví dụ 5.11: Khởi động PortA nhập,có điện trở kéo lên,PortB xuất,gía trị đầu=0x00,bộ đếm Timer1 =0x1000.

Giải:

```
#include <avr/io.h>
...
int main()
{
    DDRA=0x00    ;//PortA input
    PORTA=0xff   ;//R kéo lên PortA
    DDRB=0xff    ;//PortB output
    PORTB=0x00   ;//PortB=0x00
    TCNT1=0x1000;//khởi động đặt TCNT1=0x1000(16 bit)
    ...
}
```

5.2.6 Hoạt động của các thanh ghi đa dụng(GPRs)

C-AVR sử dụng các GPRs thay cho các biến thông số hình thức(tham số) và chúa kết quả trả về khi thực thi hàm.

- Lưu trữ các tham số(argument) trong hàm: R25 đến R8
- Chứa kết quả trả về:
 - 8 bit R24
 - 16 bit R25 - R24
 - 32 bit R22 - R25
 - 64 bit R18 - R25
- Các trường hợp dài hơn: cắt trong stack

❖ Câu hỏi ôn tập

1. C-AVR có bao nhiêu kiểu bộ nhớ?Kiểu bộ nhớ cho hằng ?và biến ?
2. C-AVR mặc định kiểu bộ nhớ của biến là kiểu gì?
3. Nêu phương pháp khai báo hằng cắt trong bộ nhớ Flash và bộ nhớ EEPROM.
4. Khai báo chuỗi ký tự “Hello! “cắt trong bộ nhớ Flash và “Xin chào! “cắt trong bộ nhớ EEPROM.
5. Khai báo biến temp là số nguyên không dấu thay đổi giá trị liên tục và biến par1 cũng là số nguyên không dấu cập nhật giá trị tùy vào ứng dụng và lưu giữ giá trị khi cắt điện.
6. Khởi động PortB với PB0,PB1 input,PB6,PB7 output.Các input có điện trở kéo lên,output ban đầu bằng 0.

5.3 Biểu thức(Expressions)và toán tử(Opertors)

Biểu thức là tập hợp các biến và hằng(danh hiệu)gọi là các toán hạng(operands)liên kết với nhau bởi các ký hiệu phép toán gọi là toán tử.Giá trị trả về của biểu thức có thể là logic,hoặc giá trị số.Biểu thức được gán cho tên biến đã được khai báo trước bởi dấu(=).Các toán tử trong biểu thức có thể là toán tử số học,logic theo bit(bitwise),quan hệ...Trong một biểu thức các toán tử được thực hiện theo thứ tự ưu tiên(precedence),nên khi thiết lập biểu thức có nhiều phép toán khác nhau,ta phải lưu ý thứ tự ưu tiên toán tử!

5.3.1 Toán tử số học

Bảng 5.3 minh họa các toán tử số học.

Bảng 5.3: Các toán tử số học

Toán tử	Ký hiệu	Ý nghĩa
Nhân	*	Nhân 2 toán hạng
Chia	/	Chia toán hạng trái cho toán hạng phải
Phần dư phép chia(Modulo)	%	Phần dư phép chia toán hạng trái cho toán hạng phải
Cộng	+	Cộng 2 toán hạng
Trừ/âm	-	Trừ toán hạng trái cho toán hạng phải/số âm

Ví dụ 5.12: Biểu diễn các phép toán số học:

```
int y ;  
char x,u;  
...  
x=10 ;  
u=0xfa ;  
y=u/x+100*x - 0x20;//y=phần nguyên (250/10)+100x10-32=993=0x03e1
```

5.3.2 Toán tử logic theo bit(bitwise)

Bảng 5.5 minh họa các toán tử theo bit.

Bảng 5.5: Các toán tử theo bit

Toán tử	Ký hiệu	Ý nghĩa
Bù 1	~	Đảo các bit của toán hạng
Dịch trái	<<	Dịch trái các bit của toán hạng số lần bên phải ký hiệu
Dịch phải	>>	Dịch phải các bit của toán hạng số lần bên phải ký hiệu
AND	&	AND các bit cùng trọng số 2 toán hạng
OR		OR các bit cùng trọng số 2 toán hạng
XOR	^	XOR các bit cùng trọng số 2 toán hạng

Ví dụ 5.13: Biểu diễn các phép toán logic theo bit, giả sử PINA=0x3c, PORTB=0xfa:

```
...  
char a,b,c;  
...  
a=PINA; //a=0x3c  
b=(a&0x0f)| 0x30; //b=(0x3c&0x0f)| 0x30=0x3c  
c=PORTB ^~(1<<5); //c=11111010 ^ ~(00100000)=11011010
```

5.3.3 Toán tử logic-quan hệ

Kết quả trả về của phép toán logic-quan hệ là giá trị logic TRUE=1 hoặc FALSE=0, thường sử dụng trong các biểu thức điều kiện.

Bảng 5.7 minh họa các toán tử logic-quan hệ.

Bảng 5.7: Các toán tử logic-quan hệ

Toán tử	Ký hiệu	Ý nghĩa
AND	&&	Trả về TRUE nếu 2 toán hạng=1, FALSE nếu có 1 toán hạng=0
OR		Trả về TRUE nếu ít nhất 1 toán hạng=1, FALSE nếu cả 2 toán hạng=0
Bằng	==	Trả về TRUE nếu 2 toán hạng bằng nhau
Không bằng(khác)	!=	Trả về TRUE nếu 2 toán hạng không bằng nhau
Nhỏ hơn	<	Trả về TRUE nếu toán hạng trái nhỏ hơn toán hạng phải
Nhỏ hơn hoặc bằng	<=	Trả về TRUE nếu toán hạng trái nhỏ hơn hoặc bằng toán hạng phải
Lớn hơn	>	Trả về TRUE nếu toán hạng trái lớn hơn toán hạng phải
Lớn hơn hoặc bằng	>=	Trả về TRUE nếu toán hạng trái lớn hơn hoặc bằng toán hạng phải

Ví dụ 5.14: Cho x=15,y=0x12,tìm kết quả trả về của các biểu thức sau:

```
x && y=1  
x & y=0x02  
x || y=1
```

```

x | y=0x1f
x==y=0
x!=y=1
x >y=0
x>=y=0
x<y=1
x<=y=1

```

5.3.4 Toán tử tăng 1,giảm 1

Toán tử tăng 1 thực hiện phép toán cộng 1 vào toán hạng: $x=x+1$.Tùy vào ký hiệu đặt trước hoặc sau toán hạng,cộng 1 sẽ được thực hiện trước hoặc sau khi thực hiện kết quả biểu thức:

```

++x      ;//x=x+1 sau đó tính x(pre-increment)
x++      ;//tính x sau đó x=x+1(post-increment)

```

Toán tử giảm 1 cũng tương tự nhưng thực hiện $x=x-1$:

```

--x      ;//x=x-1 sau đó tính x(pre-decrement)
x--      ;//tính x sau đó x=x-1(post-decrement)

```

Ví dụ 5.15: Xem các kết quả trả về của các biểu thức sau:

```

...
j=1;
k=2*++j;/k=2x2=4,j=2
j--;    //j=1
k=2*j++;/k=2x1=2,j=2

```

5.3.5 Toán tử gán kép

Trường hợp biến vừa là toán hạng vừa là giá trị biểu thức,có thể sử dụng các toán tử gán kép để đơn giản cách viết biểu thức.Bảng 5.8 liệt kê các toán tử gán kép

Bảng 5.8: Các toán tử gán kép

Ký hiệu	Mô tả	Ví dụ minh họa
$+=$	Cộng 2 toán hạng và gán kết quả vào toán hạng trái	$a+=b \rightarrow a=a+b$
$-=$	Trừ 2 toán hạng và gán kết quả vào toán hạng trái	$a-=b \rightarrow a=a-b$
$*=$	Nhân 2 toán hạng và gán kết quả vào toán hạng trái	$a*=b \rightarrow a=a*b$
$/=$	Chia 2 toán hạng và gán kết quả vào toán hạng trái	$a/=b \rightarrow a=a/b$
$\%=$	Phần dư chia 2 toán hạng và gán kết quả vào toán hạng trái	$a\%=b \rightarrow a=a \% b$
$<<=$	Dịch trái số lần bên phải ký hiệu và gán kết quả vào toán hạng trái	$a<<=3 \rightarrow a=a<<3$
$>>=$	Dịch phải số lần bên phải ký hiệu và gán kết quả vào toán hạng trái	$a>>=3 \rightarrow a=a>>3$
$\&=$	AND theo bit 2 toán hạng và gán kết quả vào toán hạng trái	$a \&=b \rightarrow a=a \& b$
$\mid=$	OR theo bit 2 toán hạng và gán kết quả vào toán hạng trái	$a \mid=b \rightarrow a=a \mid b$
$\wedge=$	EXOR theo bit 2 toán hạng và gán kết quả vào toán hạng trái	$a\wedge=b \rightarrow a=a\wedge b$

5.3.6 Các toán tử khác

Ngoài các toán tử trên,còn các toán tử khác như trong Bảng 5.9.

Bảng 5.9: Các toán tử khác

Ký hiệu	Mô tả	Ví dụ minh họa
sizeof	Trả về độ dài(số byte)của biến	<code>int x=0x1234; a=sizeof(x); //a=2</code>
$\&$	Trả về địa chỉ của biến	<code>D/c x=0x0100,x=0xdf a=&x; //a=0x0100</code>
*	Con trỏ đến biến	<code>D/c x=0x0100,x=0xdf a=&x; //a=0x0100 b=*a ; //b=0xdf</code>
(đk)?(op1):(op2)	điều kiện(đk)thỏa trả về op1, không thỏa trả về op2	<code>a=2,b=5 c=(a>=b)? 10:20; //c=20</code>

5.3.7 Mức ưu tiên và thứ tự thực hiện toán tử

Trường hợp trong một biểu thức có nhiều toán tử, ta phải xét mức và thứ tự ưu tiên toán tử để trình biên dịch trả kết quả về đúng theo thiết kế. Các toán tử xét theo mức ưu tiên trước và sau đó là thứ tự nếu được xếp cùng mức ưu tiên. Bảng 5.10 tóm tắt các mức ưu tiên theo chỉ số mức càng thấp mức ưu tiên càng cao và thứ tự ưu tiên cùng mức từ trái sang phải hay phải sang trái.

Bảng 5.10: Mức và thứ tự ưu tiên các toán tử

Tên gọi	Mức	Toán tử	Thứ tự
Sơ cấp	1	$() . [] ->$	Trái sang phải
Đơn nguyên	2	$! ~ - (type) * & ++ -- sizeof$	Phải sang trái
Nhi phân	3	$* / %$	Trái sang phải
Số học	4	$+ -$	Trái sang phải
Dịch	5	$<<>>$	Trái sang phải
Quan hệ	6	$< <= > > =$	Trái sang phải
Điều kiện bằng	7	$== !=$	Trái sang phải
Logic theo bit	8	$\&$	Trái sang phải
Logic theo bit	9	\wedge	Trái sang phải
Logic theo bit	10	$ $	Trái sang phải
Logic	11	$\&\&$	Trái sang phải
Logic	12	\parallel	Trái sang phải
Điều kiện	13	$? :$	Phải sang trái
Gán	14	$= += -= *= /= \% = <<= > > = \&= \wedge = =$	Phải sang trái

Ví dụ 5.16: Xem kết quả trả về của các biểu thức sau:

```
x=3*2>>2 ;//x=1
x=3*(2>>2) ;//x=0
x=10+6-4 ;//x=12
x=10+6*4 ;//x=34
x=4*2/3 ;//x=2
```

❖ Câu hỏi ôn tập

- Tính giá trị các biểu thức sau:
 (a) $x=10*2/4<<1$; (b) $y=25\%4>>1|0x10$; (c) $z=0xfa>>2\&~0xca$; (d) $t=0x3f^8\&9+10/3$
- Tính giá trị các biểu thức sau, cho $a=0xdf$, $b=26$, $c=0b11010101$
 (a) $x=a\&b||c$; (b) $y=a|b\&\&c$; (c) $z=(a>>1)\wedge b++ - - c$; (d) $(++a<<1)+c--$
- Tính giá trị các biểu thức sau, cho $a=0x1ff$:
 (a) $a>>=3*0x02$; (b) $a\&=0xfa\&0x0a$; (c) $a+=a<<1$; (d) $a*=(a+1)$
- Viết phát biểu đặt PB0,PB1 input có điện trở kéo lên, PB4,PB5 output có giá trị 0, không làm ảnh hưởng đến các chân port khác.
- Viết các phát biểu đảo bit ngõ ra PD3.
- Viết phát biểu chọn dữ liệu từ ngõ vào PortA nếu nó nhỏ hơn dữ liệu ngõ vào từ PortB, và ngược lại thì chọn dữ liệu ngõ vào từ PortB (xét số không dấu)

5.4 Các phát biểu điều khiển

Các phát biểu điều khiển có cấu trúc lặp vòng như while, do/while, for thực hiện lặp vòng khỏi phát biểu theo biểu thức điều kiện, hoặc có cấu trúc chuyển hướng như if/else, switch/case rẽ nhánh đến thực hiện khối phát biểu ở đoạn chương trình khác theo biểu thức điều kiện. Ngoài ra còn các phát biểu chuyển điều khiển rẽ nhánh khác như continue, break, goto có thể kết hợp với các cấu trúc trên làm đa dạng và linh động hơn trong điều khiển chương trình.

5.4.1 Vòng lặp while

❖ Cú pháp:
while(biểu thức)
 phát biểu;
 hoặc:
while(biểu thức)

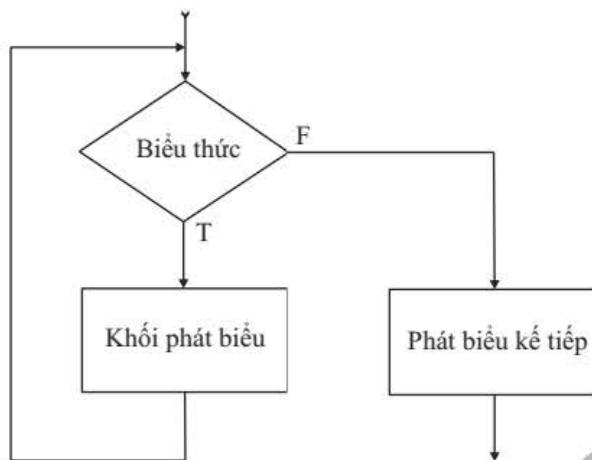
Khi thực hiện phát biểu while, biểu thức được tính giá trị:

- Nếu giá trị biểu thức trả về TRUE(T) khác 0, điều kiện thỏa mãn, sẽ thực hiện phát biểu hoặc khối phát biểu trong dấu {}
- Nếu giá trị biểu thức trả về FALSE(F)=0, điều kiện không thỏa mãn, sẽ thoát khỏi vòng lặp while thực hiện phát biểu kế tiếp

```

{
    phát biểu 1;
    phát biểu 2;
    ...
}

```



Hình 5.1: Lưu đồ phát biểu while

Ví dụ 5.17: Đoạn chương trình sau thực hiện đọc chân PC1 PortC,nếu PC1=0 tăng biến a thêm 1 và xuất ra PortB và lập vòng lại, thoát khỏi vòng lặp khi PC1=1.

Giải:

```

#include <avr/io.h>      //sử dụng các ký hiệu IORs
unsigned char a;          //khai báo biến a char 1 byte
int main()                //bắt đầu chương trình
{
    a=0                  //khởi động biến a
    DDRC=~(1<<PC1)     //PC1=0 input
    PORTC=(1<<PC1)      //R kéo lên PC1
    DDRB=0xff            //PortB output
    PORTB=a              //PortB=0x00
    while(!(PINC&(1<<PC1))) //điều kiện PC1=0 TRUE
    {
        a++;             //tăng a +1
        PORTB=a;          //xuất ra PortB
    }
    ...
}

```

Trường hợp muốn lập vòng liên tục chương trình lại từ đầu ta đặt giá trị biểu thức=1:

```

while(1)
{
    các phát biểu;
    ...
}

```

Cấu trúc trên tương đương như cấu trúc lệnh lập vòng lệnh sau trong hợp ngữ:
START:

RJMP START

Trường hợp muốn treo chương trình hay lập vòng vô hạn tại chỗ:

```
while(1);
```

Tương đương như trong hợp ngữ:

```
WAIT:    RJMP    WAIT
```

5.4.2 Vòng lặp do/while

- ❖ **Cú pháp:**
do
 phát biểu;
while(biểu thức);

Thực hiện ngược lại so với vòng lặp while,nghĩa là thực hiện khôi phát biểu trước rồi mới xét điều kiện biểu thức thỏa mãn=TRUE mới lập vòng.Do đó câu trúc do/while thực hiện ít nhất 1 vòng.

hoặc:

```
do  

{  

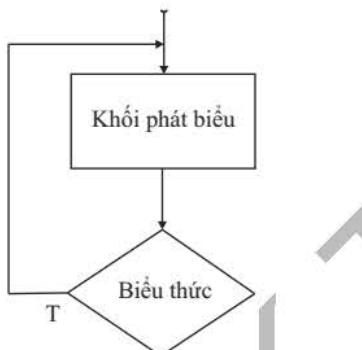
    phát biểu 1;  

    phát biểu 2;  

    ...  

}  

while(biểu thức);
```



Hình 5.2: Lưu đồ vòng lặp do/while

Ví dụ 5.18: Đoạn chương trình sau thực hiện liên tục phép chia 2 số cho đến khi thương số bằng 0.
 unsigned char x,y,u;

```
...
do  

{  

    u=x%y ;//u=dư số x/y  

    x=x/y ;//x=thương số  

}  

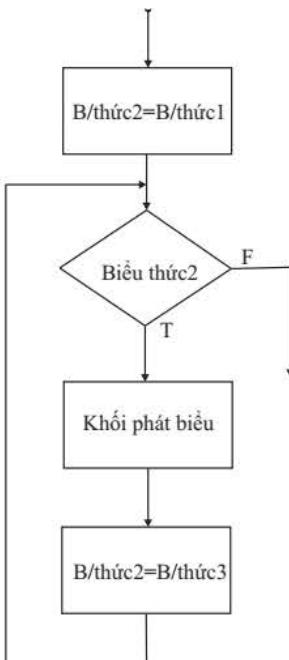
while(x!=0);
```

5.4.3 Vòng lặp for

- ❖ **Cú pháp:**
for(b/thúc1;b/thúc2;b/thúc3)
 phát biểu;
 Hoặc:
for(b/thúc1;b/thúc2;b/thúc3)
{
 phát biểu 1;
 phát biểu 2;
 ...
}

Vòng lặp for thực hiện 1 số lần lặp vòng theo điều kiện b/thúc 2:

- Khởi động lần đầu tiên xét điều kiện b/thúc 1 theo điều kiện b/thúc 2,nếu thỏa mãn(TRUE)thực hiện khôi phát biểu,nếu không thỏa mãn(FALSE)thoát khỏi vòng for
- Sau khi thực hiện xong khôi phát biểu,tính toán lại biểu thức điều kiện theo b/thúc 3 và quay lại xét tiếp theo điều kiện b/thúc 2 như bước trên



Hình 5.3: Lưu đồ vòng lặp for

Ví dụ 5.19: Thực hiện phép cộng N số đầu tiên chuỗi số sau, với giá trị N=1-255 nhập từ PortA.
 $1+3+5+7+9+\dots+(2k-1)+\dots \quad (k=1,2,3,\dots)$

Giải:

```

#include <avr/io.h>
unsigned int x,y;
unsigned char i,n;
int main()
{
    DDRA=0x00      //PortA input
    PORTA=0xff     //R kéo lên PortA
    while(1)        //lập vòng vô hạn
    {
        n=PINA;    //đọc n
        x=1;         //khởi động tổng chuỗi x=1
        y=1;         //khởi động số hạng đầu=1
        for(i=1;i<n;i++) //n vòng lặp
        {
            y=y+2;   //tính số hạng kế tiếp
            x+=y;    //tính tổng chuỗi
        }
    }
}
  
```

5.4.4 IF/ESLE

Phát biểu dạng if/else chuyển hướng hay rẽ nhánh thực hiện chương trình theo biểu thức điều kiện. Có 3 dạng phát biểu if/else:

1. Dạng IF

❖ Cú pháp:

if(biểu thức);
phát biểu;

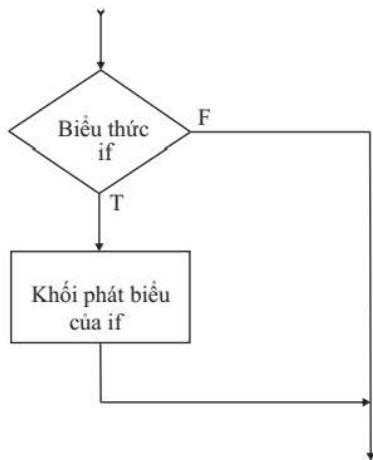
Hoặc:

if(biểu thức){

 phát biểu 1;
 phát biểu 2;

- Nếu điều kiện biểu thức thỏa mãn(TRUE), sẽ thực hiện khối phát biểu của if và tiếp tục phát biểu kế tiếp.
- Nếu điều kiện biểu thức không thỏa mãn(FALSE), sẽ bỏ qua khối phát biểu của if, thực hiện phát biểu kế tiếp.

...
}



Hình 5.4: Lưu đồ phát biểu if

Ví dụ 5.20: Thực hiện đọc dữ liệu từ PortD,nếu dữ liệu <0x20 cộng thêm 0x30 và xuất ra PortC,nếu dữ liệu >=0x20 xuất thẳng ra PortC.

Giải:

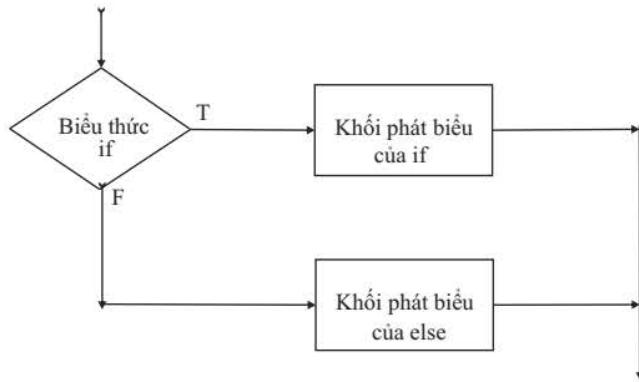
```
#include <avr/io.h>
unsigned char dat;
int main()
{
    DDRD=0x00      //PortD input
    PORTD=0xff     //R kéo lên PortD
    DDRC=0xff      //PortC output
    PORTC=0x00     //khởi động PortC=0x00
    while(1)        //lập vòng vô hạn
    {
        dat=PIND; //đọc data
        if(dat<0x20) //kiểm tra data<0x20?
            dat=dat+0x30;//nhỏ hơn cộng thêm 0x30
        PORTC=dat; //xuất ra PortC
    }
}
```

2. Dạng IF/ELSE

❖ Cú pháp:

```
if(biểu thức)
{
    khối phát biểu if;
}
else
{
    khối phát biểu else;
}
```

- Nếu điều kiện biểu thức của if =TRUE,sẽ thực hiện khối phát biểu của if ,bỏ qua khối phát biểu của else,và tiếp tục phát biểu kế tiếp sau khối phát biểu else.
- Nếu điều kiện biểu thức của if =FALSE,sẽ thực hiện khối phát biểu của else ,và tiếp tục phát biểu kế tiếp sau khối phát biểu else.



Hình 5.5: Lưu đồ phát biểu if/else

Ví dụ 5.21: Thực hiện đọc dữ liệu từ PortD,nếu dữ liệu=\$20 - \$7f xuất ra PortC,nếu ngoài tầm trên xuất ra PortC \$ff .

Giải:

```

#include <avr/io.h>
unsigned char dat;
int main()
{
    DDRD=0x00      //PortD input
    PORTD=0xff     //R kéo lên PortD
    DDRC=0xff      //PortC output
    PORTC=0x00     //khởi động PortC=0x00
    while(1)        //lặp vòng vô hạn
    {
        dat=PIND   //đọc data
        if(0x20<=dat&&dat<=0x7f) //kiểm tra data=$20-$7f?
            PORTC=dat //trong tầm xuất ra PortC
        else          //ngoài tầm
            PORTC=0xff;//xuất $ff ra PortC
        ...
    }
}

```

3. Dạng IF/ELSE IF/...

❖ Cú pháp:

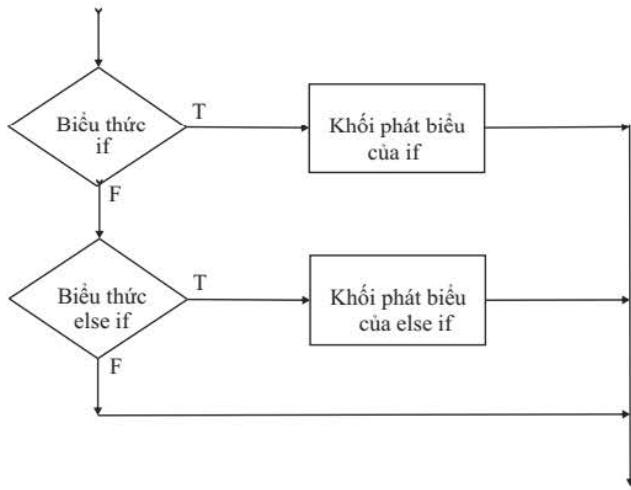
```

if(biểu thức)
{
    khối phát biểu if;
}
else if(biểu thức)
{
    khối phát biểu else;
}

```

- Nếu điều kiện biểu thức của if =TRUE,sẽ thực hiện khối phát biểu của if ,bỏ qua khối phát biểu của else,và tiếp tục phát biểu kế tiếp sau khối phát biểu else.Nếu điều kiện biểu thức của if =FALSE sẽ xét điều kiện biểu thức của else if.

- Nếu điều kiện biểu thức của else if =TRUE,sẽ thực hiện khối phát biểu của else ,và tiếp tục phát biểu kế tiếp sau khối phát biểu else. Nếu điều kiện biểu thức của else=FALSE,bỏ qua khối phát biểu của else,và thực hiện phát biểu kế tiếp khối phát biểu của else.



Hình 5.6: Lưu đồ phát biểu if/else if...

Ta có thể tiếp tục lồng thêm các phát biểu if/else if... vào các dạng cấu trúc if ở trên để tạo nhiều điều kiện rẽ nhánh khác nhau!

Ví dụ 5.22: Thực hiện đọc dữ liệu từ PortD, nếu dữ liệu=\$20 - \$7f xuất ra PortC, nếu ngoài tầm trên xuất ra PortC \$00 nếu dữ liệu <\$20 hoặc \$ff nếu dữ liệu >\$7f.

Giải:

```

#include <avr/io.h>
unsigned char dat;
int main()
{
    DDRD=0x00      ;//PortD input
    PORTD=0xff     ;//R kéo lên PortD
    DDRC=0xff      ;//PortC output
    PORTC=0x00     ;//khởi động PortC=0x00
    while(1)        //lập vòng vô hạn
    {
        dat=PIND  ;//đọc data
        if(0x20<=dat&&dat<=0x7f) //kiểm tra data=$20-$7f?
            PORTC=dat ;//trong tầm xuất ra PortC
        else if(dat>0x7f)      //data>$7f
            PORTC=0xff; //xuất $ff ra PortC
        else                //data <$20
            PORTC=0x00; //xuất $00 ra PortC
        ...
    }
}

```

5.4.5 SWITCH/CASE

Phát biểu switch/case là phát biểu chọn rẽ nhánh chương trình nhiều hướng khác nhau tùy thuộc vào giá trị của biểu thức.

❖ **Cú pháp:**

```

switch(biểu thức)
{
    case const1:
        phát biểu 1.1;
        phát biểu 1.2;
        ...
        break;
    case const2:
        phát biểu 2.1;
        phát biểu 2.2;
        ...
}

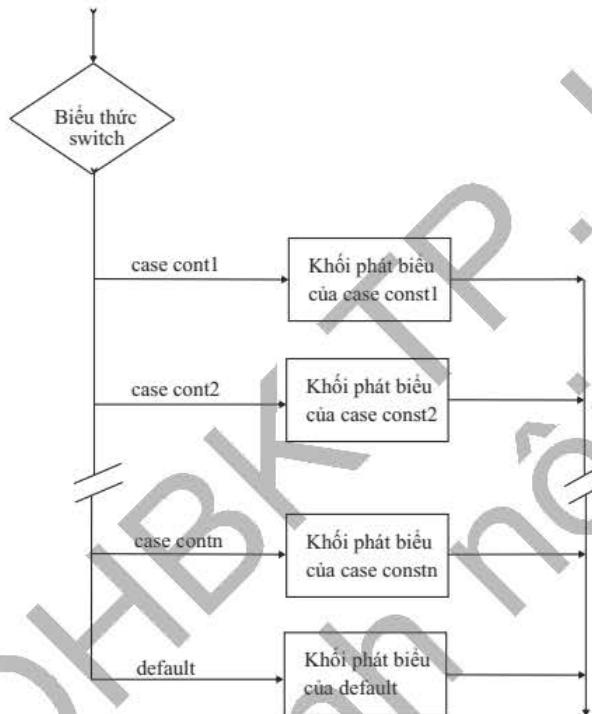
```

- Giá trị biểu thức trả về phải là các hằng số nguyên hoặc ký tự
- Các giá trị const1,const2,...constn là hằng số nguyên hoặc ký tự
- Nếu giá trị biểu thức=const1,sẽ thực hiện khối phát biểu thuộc case const1 cho đến phát biểu break sẽ thoát khỏi switch/case và thực hiện phát biểu kế tiếp sau phát biểu switch/case
- Tương tự cho trường hợp giá trị biểu thức=const2 đến constn
- Nếu giá trị biểu thức khác các giá trị case,sẽ thực hiện khối phát biểu thuộc default,và thoát khỏi switch/case đến phát biểu kế tiếp.Tùy yêu cầu cụ thể có thể không cần trường hợp default.
- Nếu không có phát biểu break, chương trình sẽ tiếp tục thực hiện khối phát biểu case kế tiếp sau đó!

```

        break;
case constn:
    phát biểu n.1;
    phát biểu n.2;
    ...
    break;
default:
    phát biểu df1
    phát biểu df2;
    ...
}

```



Hình 5.7: Lưu đồ phát biểu switch/case/break

Ví dụ 5.23: Viết một chương trình nhập một số từ PortA đặt là a và 6 bit thấp PortB đặt là b. Xét các trường hợp PB7:PB6 để thực hiện các công việc sau:

- PB 7:PB6=00: a-b xuất kết quả byte cao ra PortD, byte thấp ra PortC
- PB 7:PB6=01: a+b xuất kết quả byte cao ra PortD, byte thấp ra PortC
- PB 7:PB6=10: a*b xuất kết quả byte cao ra PortD, byte thấp ra PortC
- PB 7:PB6=11: a/b xuất thương số ra PortC, dư số ra PortD

Giải:

```

#include <avr/io.h>
unsigned char a,b,c;
unsigned int d;
int main()
{
    DDRA=0X00 //PortA input
    PORTA=0xff ;//R kéo lên PortA
    DDRB=0X00 //PortB input
    PORTB=0xff ;//R kéo lên PortB
    DDRC=0xff //PortC output
    PORTC=0x00 ;//khởi động PortC=0x00
    DDRD=0xff //PortD output
    PORTD=0x00 ;//khởi động PortD=0x00
    while(1) //lặp vòng vô hạn
    {

```

```

a=PIN A //đọc PortA
b=PIN B&0x3f;//đọc PortB che 6 bit thấp
c=PIN B>>6 ;//đọc PortB dịch PB7,PB6 xuống vị trí LSB
switch(c)
{
    case 0:           //c=0 phép trừ
        d= a - b;
        PORTC=d ;   //PortC= byte thấp d
        PORTD= d>>8 ; //dịch byte cao d xuống byte thấp xuất ra PortD
        break;         //thoát khỏi switch
    case 1:           //c=1 phép cộng
        d=a + b ;
        PORTC=d ;
        PORTD= d>>8 ;
        break;
    case 2:           //c=2 phép nhân
        d= a * b;
        PORTC=d ;
        PORTD= d>>8 ;
        break;
    case 3:           //c=3 phép chia
        PORTC=a/b ; //PortC=thương số a/b
        PORTD= a % b ;//dư số a/b
        break;
}
}
}

```

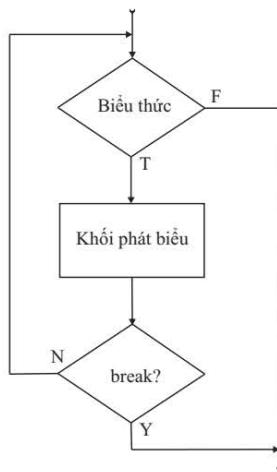
Trong chương trình trên, phép toán trừ, cộng, nhân có thể bị tràn nên sử dụng biến d unsigned int 16 bit. Khi gán d vào các thanh ghi Port 8 bit, mặc định chỉ gán byte thấp của d. Để gán byte cao của d, ta phải dịch phái xuống byte thấp trước khi gán. Trường hợp phép chia 2 số 8 bit kết quả chắc chắn là 8 bit, nên có thể gán trực tiếp biểu thức ra Port.

5.4.6 BREAK,CONTINUE,GOTO

Các phát biểu break, continue, goto có thể sử dụng kết hợp với các phát biểu while, do/while, for, switch/case để tăng thêm tính đa dạng và linh động trong các phát biểu điều khiển.

1. BREAK

Phát biểu break kết thúc khỏi phát biểu trong vòng lặp và thoát khỏi vòng lặp. Trong trường hợp có nhiều vòng lặp bao lồng nhau, phát biểu break kết thúc khỏi phát biểu và thoát khỏi vòng lặp bao trực tiếp nó. Trong phát biểu switch/case ở mục trên, phát biểu break được đặt cuối mỗi khối phát biểu của từng case để kết thúc phát biểu switch sau khi thực hiện xong khối phát biểu của case được chọn.



N=không có phát biểu break
Y=có phát biểu break

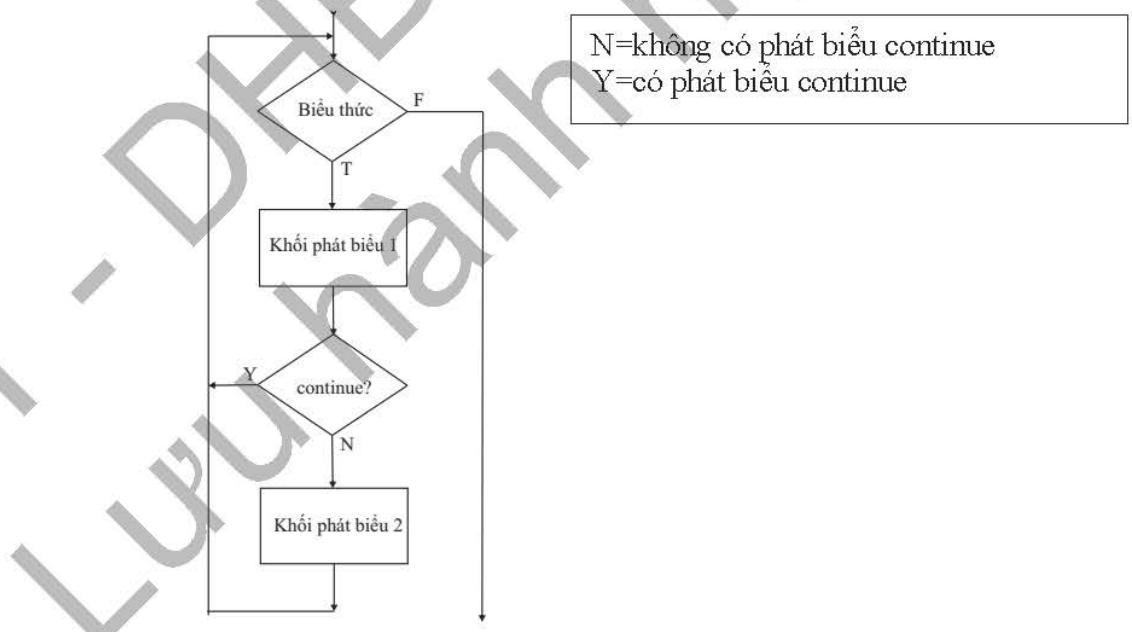
Hình 5.8: Lưu đồ thực hiện phát biểu break

Ví dụ 5.24: Đoạn chương trình sau nhận dạng PB3=0 trong số lần lặp vòng nhất định và thoát khỏi vòng lặp khi PB3=0, xuất giá trị đếm số lần lặp vòng ra PortC.

```
#include <avr/io.h>
char i;
int main()
{
    DDRB=~(1<<PB3) ;//PB3 input
    PORTB=(1<<PB3) ;//R kéo lên PB3
    DDRC=0xff ;//PortC output
    PORTC=0x00 ;//khởi động PortC=0x00
    i=0; //biến đếm
    while(i<=50) //giới hạn đếm đến 50
    {
        i++; //tăng 1 biến đếm
        if(!(PINB&(1<<PB3))) //nhận dạng PB3=0?
            break ; //thoát
    }
    PORTC=i ; //xuất biến đếm ra PortC
    while(1); //lặp vòng tại chỗ
}
```

2. CONTINUE

Phát biểu continue giống phát biểu break ở chỗ kết thúc khối phát biểu trong vòng lặp và thoát khỏi vòng lặp. Tuy nhiên phát biểu continue sẽ tiếp tục thực hiện vòng lặp lại từ đầu. Thông thường phát biểu continue được sử dụng để loại bỏ khối phát biểu đứng sau nó trong vòng lặp nhưng vẫn tiếp tục thực hiện vòng lặp cho đến khi biểu thức điều kiện không thỏa mãn.



Hình 5.9: Lưu đồ thực hiện phát biểu continue

Ví dụ 5.25: Xem lại đoạn chương trình sau và so sánh với ví dụ 5.24.

```
#include <avr/io.h>
char i;
int main()
{
    DDRB=~(1<<PB3) ;//PB3 input
    PORTB=(1<<PB3) ;//R kéo lên PB3
    DDRC=0xff ;//PortC output
    PORTC=0x00 ;//khởi động PortC=0x00
```

```

i=0;           //biến đếm
while(i<=50)   //giới hạn đếm đến 50
{
    i++;        //tăng 1 biến đếm
    if(!(PINB&(1<<PB3))) //nhận dạng PB3=0?
    {
        PORTC=i ;//xuất biến đếm ra PortC
        continue ;//thoát,lập vòng lại
    }
    PORTC^=0x01 ;//đảo bit PC0
}
while(1);      //lặp vòng tại chỗ
}

```

- Trong ví dụ 5.24,khi số lần lặp vòng $i \leq 50$,nếu $PB3=0$,chương trình sẽ xuất ra $PortC=i$ và thoát khỏi vòng lặp.
- Trong ví dụ 5.25,khi số lần lặp vòng $i \leq 50$,nếu $PB3=1$ thực hiện đảo bit PC0,nếu $PB3=0$ sẽ xuất ra $PortC=i$ và trở về tiếp tục vòng lặp.Chương trình chỉ thoát khỏi vòng lặp khi $i > 50$.

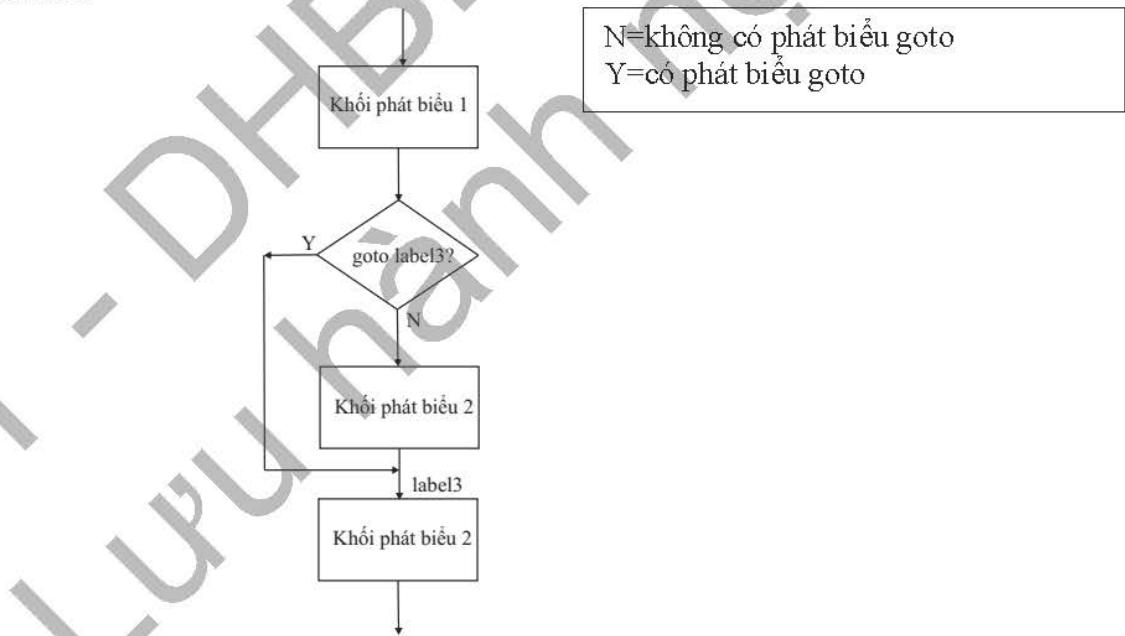
3. GOTO

Phát biểu goto tương tự như lệnh JMP trong hợp ngữ,điều khiển chương trình rẽ nhánh không điều kiện đến vị trí phát biểu có gán nhãn tương ứng.

❖ Cú pháp:

goto Label

- **Label:** nhãn có quy định như danh hiệu và phải có dấu (:) đứng cuối,đặt ở đầu dòng vị trí phát biểu là đích đến.



Hình 5.10: Lưu đồ thực hiện phát biểu goto

Ví dụ 5.26: Viết lại ví dụ 5.24 làm việc như ví dụ 5.25,sử dụng phát biểu goto.

Giải:

```

#include <avr/io.h>
char i;
int main()
{
    DDRB=~(1<<PB3) ;//PB3 input
    PORTB=(1<<PB3) ;//R kéo lên PB3
    DDRC=0xff ;//PortC output
    PORTC=0x00 ;//khởi động PortC=0x00
}

```

```

i=0;           //biến đếm
START: while(i<=50)    //giới hạn đếm đến 50
{
    i++;        //tăng 1 biến đếm
    if(!(PINB&(1<<PB3))) //nhận dạng PB3=0?
    {
        PORTC=i ;//xuất biến đếm ra PortC
        goto START;//nhảy về START
    }
    PORTC^=0x01 ;//đảo bit PC0
}
while(1);      //lặp vòng tại chỗ
}

```

❖ Câu hỏi ôn tập

- Viết đoạn chương trình nhập số từ PortA,nếu là số âm bù 2 số nhập.
- Viết đoạn chương trình nhập số từ PortA,nếu là số lẻ xuất ra PortB,là số chẵn xuất ra PortC.
- Viết đoạn phát biểu kiểm tra chân PC1 liên tục và thoát nếu PC1=0.
- Viết đoạn phát biểu kiểm tra chân PC1,PC0 liên tục và thoát nếu có 1 ngõ=0.
- Viết đoạn chương trình đọc ngõ vào PC1,PC0 và thực hiện như sau:
PC1:PC0=0 đặt PB0=1,PC1:PC0=1 đặt PB1=1,PC1:PC0=2 đặt PB2=1,PC1:PC0=3 đặt PB3=1.
- Viết một đoạn chương trình đọc giá trị từ PortD tối đa 50 lần,và thoát khi đọc được giá trị \$0d.

5.5 Hàm(Functions)

Hàm tương tự như chương trình con trong hợp ngữ,được định nghĩa 1 lần duy nhất thực hiện 1 hoặc một số tác vụ lặp đi lặp lại của chương trình.Khi có yêu cầu thực hiện các tác vụ này,chuong trinh chính chỉ cần gọi tên hàm và có thể gọi nhiều lần.Hàm trong C linh động hơn do chỉ cần khai báo tổng quát các thông số hình thức hay tham số(arguments).Khi gọi hàm chương trình chính sẽ thay các thông số hình thức bằng biến hoặc hằng cụ thể.

Hàm có thể được viết thành thư viện để có thể sử dụng trong nhiều chương trình khác nhau bằng khai báo #include.C-AVR có rất nhiều hàm thư viện chức năng,một trong các hàm ta thường sử dụng nhất là hàm định nghĩa các ký hiệu IORs luôn có mặt ở dòng đầu chương trình:

```
#include <avr/io.h>
```

Định dạng cơ bản một cấu trúc hàm như sau:

```

type Tên_hàm(type par1,type par2,...)
{
    thân hàm;
    ...
    [return biểu thức]
}
```

- type: kiểu dữ liệu kết quả hàm trả về thông thường là char,int,long,float.Trường hợp hàm không cần trả về kết quả,kiểu dữ liệu là void.

- Tên_hàm: quy định như danh hiệu dùng để gọi hàm

- (type par1,type par2,...): par1,par2,...là danh sách các thông số hình thức sử dụng trong hàm type là kiểu dữ liệu của từng thông số tương ứng

Trường hợp hàm không có thông số hình thức đặt (void) hay ()

- Thân hàm tương tự như thân chương trình chính, gồm khai báo các biến và hằng sử dụng riêng cho hàm và các phát biểu,được đặt trong dấu({...})

- Kết thúc hàm là phát biểu return và biểu thức cần trả về giá trị.Trường hợp không cần giá trị trả về có thể viết return 0 hay bỏ qua phát biểu return

Để gọi hàm trong chương trình chính,chỉ cần gọi tên hàm và gán giá trị cụ thể cho các thông số.

Ví dụ 5.27: Xem lại hàm tạo trễ giới thiệu ở ví dụ 5.1

```

void delay_ms(unsigned int n)
{

```

```

unsigned int i;
for(i=0;i<=n;i++)// Td=6xn MC
}
...
int main()
{
...
delay_ms(13333);
...
}

```

- Hàm này chỉ tạo thời gian trễ không cần giá trị trả về nên có kiểu void
- Tên hàm là delay_ms thể hiện chức năng tạo trễ tính bằng ms
- Thông số hình thức n kiểu unsigned int: $n = 0 \div 65535$
- Trong thân hàm khai báo biến cục bộ i và phát biểu vòng lặp for tạo trễ, giới hạn số lần lập vòng chính là thông số n sẽ được gán giá trị cụ thể khi gọi hàm.
- Hàm này không cần giá trị trả về nên bỏ qua phát biểu return
- Thời gian trễ tính theo hàm là $Td(\mu s) \approx 6nxMC$ (tham khảo file.lss sau khi biên dịch). Ví dụ với $Fosc=8Mhz, 1MC=0.125\mu s$, muốn tạo trễ $Td=10ms \rightarrow n=10000/(6x0.125) \approx 13333$.
- Trong chương trình chính, muốn tạo trễ 10ms chỉ cần gọi hàm và thay n=13333:
`delay_ms(13333);`

5.5.1 Khai báo và định nghĩa hàm

Để việc gọi hàm được chấp nhận, ta phải khai báo và định nghĩa hàm. Có hai cách như sau:

1. Không cần khai báo hàm, định nghĩa trực tiếp hàm ở đầu chương trình, sau phần khai báo biến toàn cục và trước khi bắt đầu chương trình chính. Ví dụ như sau:

```

#include <avr/io.h>
unsigned char a,b,c;
void delay_ms(unsigned int n)
{
    unsigned int i;
    for(i=0;i<=n;i++)// Td=6xn MC
}
int main()
{
...
delay_ms(13333);
...
}

```

2. Khai báo trước tên hàm ở vị trí đầu chương trình như trên, định nghĩa hàm sau chương trình chính.

Cách này thường sử dụng khi ta cần hoàn thiện trước ý tưởng giải thuật chương trình chính, bổ sung định nghĩa hàm sau, hoặc định nghĩa các hàm bao gồm các hàm khác bên trong. Ví dụ như sau:

```

#include <avr/io.h>
unsigned char a,b,c;
void delay_ms(unsigned int n);//khai báo hàm delay_ms
...
int main()
{
...
delay_ms(13333);
...
}
void delay_ms(unsigned int n) //định nghĩa hàm delay_ms
{
    unsigned int i;

```

```
for(i=0;i<=n;i++)// Td=6xn MC  
}
```

5.5.2 Gọi hàm-Các giá trị trong hàm

Sau khi khai báo hàm, ta có thể gọi hàm ở bất kỳ thời điểm và vị trí nào trong chương trình và trong hàm khác nếu có yêu cầu (tất nhiên hàm phải được định nghĩa). Để gọi hàm ta chỉ cần viết tên hàm và thay các thông số hình thức bằng các giá trị biến hay hằng cụ thể cùng kiểu dữ liệu đã khai báo cho thông số hình thức. Tên hàm có thể là một thành phần toán hạng. Ta cần phân biệt rõ các giá trị sử dụng trong hàm để có kết quả trả về đúng yêu cầu khi thực hiện hàm.

1. Biến toàn cục

- Là các biến được khai báo trong chương trình chính, hoặc ngoài hàm
- Trong hàm khi sử dụng các biến toàn cục, khi trả kết quả về biến toàn cục sẽ lưu giá trị cuối cùng được tính toán trong hàm. Giá trị này chỉ phụ thuộc vào kiểu dữ liệu của biến toàn cục không phụ thuộc vào kiểu dữ liệu của hàm.

2. Biến cục bộ

- Là các biến được định nghĩa trong hàm và chỉ được sử dụng trong hàm đã định nghĩa.
- Khi thoát khỏi hàm, giá trị các biến này nói chung sẽ bị xóa, trừ trường hợp có phát biểu return yêu cầu trả kết quả giá trị của biến.
- Các biến cục bộ của các hàm khác nhau có thể đặt tên trùng nhau, nhưng trình biên dịch vẫn phân biệt sử dụng biến theo hàm đã được khai báo. Trường hợp biến cục bộ trùng tên biến toàn cục, sẽ ưu tiên chọn biến cục bộ trong hàm.

❖ Phân loại lưu trữ các giá trị biến cục bộ

Biến cục bộ được phân thành 3 loại lưu trữ khác nhau:

- **Tự động:** khai báo **auto type tên biến**, có thể bỏ qua từ khóa auto

Ví dụ: auto char x; hay char x;

Loại biến này có giá trị không xác định khi khởi động, nên người lập trình cần khởi động giá trị cho nó. Biến loại auto tự động bị xóa khi thoát khỏi hàm, nghĩa là ô nhớ cho nó sẽ bị mất và nó không còn giá trị khi thoát khỏi hàm.

- **Tĩnh:** khai báo **static type tên biến**

Ví dụ: static int x;

Biến loại static có giá trị khởi động bằng 0 lần đầu tiên gọi hàm, và vẫn lưu giá trị khi thoát khỏi hàm. Như vậy từ lần gọi hàm kế tiếp biến static sẽ có giá trị khởi động bằng giá trị lần thoát khỏi hàm trước đó.

- **Thanh ghi:** khai báo **register type tên biến**

Biến loại thanh ghi tương tự như biến loại auto, nhưng khác ở chỗ biến thanh ghi sẽ thực thi chương trình nhanh hơn. Việc gán loại biến thanh ghi tùy thuộc vào tổ chức bộ nhớ/thanh ghi của mỗi loại MCU.

- C-AVR thường mặc định gán các GPRs cho các biến cục bộ, hoạt động các thanh ghi như đã trình bày ở mục 5.2.6.

3. Thông số hình thức

- Là các danh hiệu đại diện cho biến/hằng được sử dụng trong hàm, được gọi là các thông số ngõ vào của hàm.
- Khi khai báo hàm phải khai báo danh sách tất cả các thông số hình thức và kiểu dữ liệu của từng thông số.
- Trong phần thân hàm, các thông số hình thức tương tự như các biến và hằng có mặt trong các phát biểu.
- Khi gọi hàm, ta phải thay các thông số hình thức bằng các biến/hằng cụ thể đã định nghĩa và có cùng kiểu dữ liệu đã khai báo cho từng thông số hình thức.
- Trình biên dịch sẽ thay thông số hình thức bằng biến/hằng đã gán theo từng dòng một trong thân hàm khi biên dịch.
- Các biến được gán cho các thông số hình thức vẫn giữ nguyên giá trị khi thoát khỏi hàm
- Các thông số hình thức làm việc tương tự như các biến cục bộ, sẽ bị xóa khi thoát khỏi hàm

4. Giá trị trả về

- Trường hợp cần kết quả giá trị trả về từ hàm, ta phải có phát biểu cuối cùng trong hàm là: **return biểu thức;**
- Biểu thức sau return có cùng kiểu dữ liệu đã khai báo trước tên hàm
- Chỉ duy nhất có 1 giá trị trả về từ hàm. Như vậy nếu muốn có từ 2 giá trị trả về trả lén, ta phải sử dụng thêm biến toàn cục.
- Trường hợp không cần giá trị trả về từ hàm, khai báo kiểu void trước tên hàm, kết thúc hàm bằng phát biểu:
return 0;
hay:
return;
hay có thể bỏ qua phát biểu return.

Ví dụ 5.28: Viết 1 chương trình nhập 1 số 8 bit vào PortB và PortA, nhân chúng với nhau và chuyển kết quả nhân thành số BCD 5 digit dạng nén đặt trong 3 biến.

Giải:

Ta đã biết phương pháp chuyển số nhị phân sang BCD bằng cách liên tục chia nó cho 10, số dư đầu tiên là digit0 và số dư cuối cùng là digit cao nhất. Kết quả phép nhân 2 số 8 bit dài 16 bit nên có tối đa 5 digit.

Trong chương trình dưới đây, ta sẽ sử dụng 3 hàm gồm:

- Hàm nhân 2 số nhị phân 8 bit trả về kết quả 16 bit là biến toàn cục bộ f.
- Hàm chia số nhị phân 16 bit cho 10, trả về thương số 16 bit và số dư 8 bit. Do phải trả về 2 giá trị nên ta chọn giá trị trả về là biến toàn cục bộ z, và biến toàn cục u chứa số dư phép chia.
- Hàm chuyển đổi từ số nhị phân 16 bit sang BCD không cần trả về giá trị. Trong hàm các digit bcd được lưu vào các biến toàn cục: u=digit4, b=digit3:digit2, a=digit1:digit0.

//-----

```
#include <avr/io.h>
#define uchar unsigned char //định nghĩa uchar
#define uint unsigned int //định nghĩa uint
uchar a,b,u; //các biến toàn cục
uint bin16;
uint mul16(uchar x,uchar y); //khai báo hàm nhân 2 số nhị phân 8 bit, trả về 16 bit
uint div16_10(uint z); //khai báo hàm chia số nhị phân 16 bit cho 10, trả về 16 bit
void bin16_bcd5(uint t); //khai báo hàm chuyển đổi số nhị phân 16 bit sang bcd 5 digit
int main()
{
    DDRA=0x00 ; //PortA input
    PORTA=0xff ; //R kéo lên PortA
    DDRB=0x00 ; //PortB input
    PORTB=0xff ; //R kéo lên PortB
    while(1) //lặp vòng vô hạn
    {
        a=PINA ; //đọc data từ PortA
        b=PINB ; //đọc data từ PortB
        bin16=mul16(a,b); //gọi hàm nhân 2 số nhị phân 8 bit
        bin16_bcd5(bin16); //gọi hàm chuyển đổi số nhị phân 16 bit sang bcd 5 digit
    }
}
//-----
uint mul16(uchar x,uchar y) //định nghĩa hàm nhân 2 số nhị phân 8 bit
{
    uint f ;//khai báo biến cục bộ
    f=x*y ;//tính phép nhân
    return f ;//trả về kết quả nhân
}
//-----
void bin16_bcd5(uint t) //khai báo hàm chuyển đổi số nhị phân 16 bit sang bcd 5 digit
{
```

```

        uint tam;           //khai báo biến cục bộ
        tam=div16_10(t);   //gọi hàm chia 10
        a=u ;              //gán digit0 vào a
        tam=div16_10(tam); //gọi hàm chia 10
        a|=(u<<4);       //gán digit1 vào 4 bit cao a
        tam=div16_10(tam); //gọi hàm chia 10
        b=u ;              //gán digit2 vào b
        tam=div16_10(tam); //gọi hàm chia 10
        b|=(u<<4);       //gán digit3 vào 4 bit cao b
        tam=div16_10(tam); //gọi hàm chia 10,u=digit4
    }
}

uint div16_10(uint z)      //định nghĩa hàm chia số nhị phân 16 bit cho 10
{
    u=z%10;//u=dư số phép chia
    z=z/10 ;//gán z=thương số
    return z ;//trả về thương số
}

```

5.5.3 Sự đệ quy(Recursion)

Một hàm đệ quy là trong hàm gọi chính nó.Một trong các thế mạnh của ngôn ngữ C là có thể thực hiện hàm đệ quy.Do đó những bài toán có tính lặp lại theo dạng chuỗi,hoặc bước tính giống nhau sử dụng hàm đệ quy rất thuận lợi!

Tuy nhiên ta phải hết sức lưu ý khi sử dụng hàm đệ quy,nếu không kiểm soát tốt,vùng nhớ sẽ bị tràn và chương trình sẽ bị rối loạn!Khi một hàm được gọi,các biến cục bộ,hàng và địa chỉ được cất trong vùng stack và heap được phân bổ(vùng heap hoạt động tương tự như vùng stack ,chứa các biến con trả định vị gián tiếp).Khi thoát khỏi hàm,vùng nhớ này sẽ bị xóa.Trong trường hợp đệ quy,mỗi lần hàm được gọi nữa,một vùng nhớ mới tương tự như vậy được phân bổ để cất các biến,hàng,địa chỉ tương ứng...Do đó nếu gọi hàm lặp lại liên tục đến lúc nào đó vùng nhớ sẽ bị tràn!

Ví dụ sau minh họa chi tiết hơn về hàm đệ quy.

Ví dụ 5.29: Viết chương trình tính giai thừa.

Giải:

Công thức tổng quát tính n!:

$n!=n.(n-1).(n-2)\dots2.1$

$0!=1$

Do đó ta sẽ dùng hàm đệ quy tính n! bằng cách lấy $n.(n-1)!$ cho đến khi $n=0$ là kết thúc!

```

//include <avr/io.h>
int n;
int fact(int m)
{
    if(m==0)
        return 1;
    else
        return (m*fact(m-1));
}

int main(void)
{
    n=fact(5);
    while (1);
}

```

- Kết quả $n=5\times4\times3\times2\times1=120$

- Theo ví dụ trên,mỗi lần thực hiện hàm fact cần phải lưu 6 byte gồm:

- biến cục bộ int m 2 byte

- giá trị trả về int 2 byte
- địa chỉ trả về 2 byte

Với 5! cần $5 \times 6 = 30$ byte ô nhớ.Nếu n lén số ô nhớ cần lưu sẽ tăng theo!

❖ Câu hỏi ôn tập

1. Viết một hàm trả về giá trị một biến 2 byte ghép bởi 2 biến 1 byte.
2. Viết một hàm nhân 2 số không dấu 16 bit.
3. Viết một hàm trả về thương số và dư số phép chia 2 số nhị phân 16 bit.
4. Viết hàm tính $x^2 + x + 1$.
5. Viết một hàm đọc byte dữ liệu từ PortA chuyển sang số BCD không nén 3 digit lần lượt cất trong các biến a,a+1,a+2.

5.6 Con trỏ(Pointers)

Con trỏ là một kiểu dữ liệu đặc biệt dùng để truy xuất giá trị của biến thông qua địa chỉ của nó, tương tự như phương pháp định vị gián tiếp bộ nhớ. Con trỏ có thể ẩn định địa chỉ cụ thể của biến trong bộ nhớ và làm thông số hình thức trong hàm,nên thường được sử dụng rất hiệu quả trong lập trình C,nhất là các tác vụ liên quan đến phần cứng trong giao tiếp bộ nhớ,ngoại vi.

❖ Cú pháp:

type *tên_con_trỏ;

- type: kiểu dữ liệu của biến mà con trỏ truy xuất thường là char,int
- Dấu (*) ký hiệu biến con trỏ
- tên_con_trỏ: tương tự như tên biến,gọi là biến con trỏ

Ví dụ: `char *p; // p là biến con trỏ truy xuất biến có kiểu dữ liệu char 1 byte`
`int *pf; // pf là biến con trỏ truy xuất biến có kiểu dữ liệu int 2 byte`

➤ C-AVR mặc định biến con trỏ kiểu dữ liệu int 2 byte thuộc kiểu bộ nhớ data(SRAM)

5.6.1 Con trỏ truy xuất giá trị biến

Để truy xuất giá trị biến,trước tiên phải gán địa chỉ của biến cho con trỏ,sử dụng ký hiệu (&):

p=#//gán địa chỉ biến num cho con trỏ p

Sau đó đọc giá trị biến qua biến con trỏ trỏ vào địa chỉ của biến:

a=*p;//đọc biến num gán vào a,a=num

Ta xem ví dụ sau và chạy mô phỏng trên Atmel Studio 7.

Ví dụ 5.30:

```
#include <avr/io.h>
char num1,a,*p1;
int num2,b,*p2;
int main(void)
{
    while (1)
    {
        num1=0xfa; //num1=0xfa
        num2=num1*100;//num2=0x61a8
        p1=&num1; //p1=địa chỉ num1
        a=*p1; //a=(p1)=num1
        p2=&num2; //p2=địa chỉ num2
        b=*p2; //b=(p2)=num2
    }
}
```

Sau khi chạy chương trình mô phỏng ta được:

- Địa chỉ num1=0x108→num1(0x108)=0xfa
- Địa chỉ num2=0x107:0x106→num2(0x107:0x106)=0x61a8
- p1=&num1=0x108
- p2=&num2=0x107:0x106
- a=*p1=(0x108)=0xfa
- b=*p2=(0x107:0x106)=0x61a8

❖ Một số biểu thức con trỏ đặc biệt

1. Truy xuất con trỏ lồng trong con trỏ

Ví dụ như sau:

```
int i,j,*p1,*p2,*p3;  
p1=&i ;//p1 trỏ địa chỉ i  
p2=&p1 ;//p2 trỏ địa chỉ p1  
p3=&p2 ;//p3 trỏ địa chỉ p2  
j=***p3 ;//j=((p3))
```

Phát biểu trên tương đương:
j=i ;

2. Tăng/giảm biến con trỏ

Ví dụ như sau:

```
int *p1;  
long *p2;  
p1++; //giá trị p1 sẽ tăng 2  
p2--; //giá trị p2 sẽ giảm 4
```

3. Thực hiện toán tử

Ví dụ như sau:

```
char a,*p;  
a=*p++; //gán vào a giá trị biến có địa chỉ trỏ bởi p,sau đó tăng địa chỉ p  
a=++p; //tăng địa chỉ p,sau đó gán vào a giá trị biến có địa chỉ trỏ bởi p  
a=++*p; //tăng giá trị biến có địa chỉ trỏ bởi p gán vào a, giữ nguyên địa chỉ p  
a=(*p)++; //gán vào a giá trị biến trỏ bởi p,sau đó tăng giá trị biến trỏ bởi p, giữ nguyên địa chỉ p
```

5.6.2 Con trỏ ẩn định địa chỉ biến

Thông thường trình biên dịch tự động chọn địa chỉ cho biến ta không thể xác định được, ta chỉ có thể biết địa chỉ biến khi xem file.lss hoặc khi chạy mô phỏng. Tuy nhiên sử dụng con trỏ ta có thể ẩn định địa chỉ cụ thể cho biến và truy xuất dễ dàng biến trong vùng nhớ ẩn định.

Ta xem ví dụ minh họa sau.

Ví dụ 5.31:

```
#include <avr/io.h>  
char i,n,x,*p;  
const int S_buf=0x200;//S_buf=địa chỉ đầu vùng nhớ SRAM  
int main()  
{  
    n=0x30;//giá trị đầu chuỗi  
    p=S_buf;//địa chỉ đầu vùng nhớ  
    for(i=0;i<=9;i++)//lặp vòng 10 lần  
        *p++=n+i; //gán vào địa chỉ trỏ bởi p giá trị n+i,tăng địa chỉ p  
    p=S_buf+5;//p trỏ địa chỉ 0x205  
    x=*p;//x=(0x205)=0x35  
    while(1);  
}
```

Trong ví dụ trên, ta sử dụng con trỏ khởi động và nạp các giá trị biến vào vùng nhớ SRAM địa chỉ đầu S_buf=0x200. Vòng lặp for lặp lượt gán 10 giá trị vào vùng nhớ:

(0x200)=0x30, (0x201)=0x31 ..., (0x209)=0x39

Sau đó để truy xuất giá trị biến tại địa chỉ 0x205, chỉ cần đặt con trỏ p=S_buf+5 và gán x=*p.

5.6.3 Con trỏ làm thông số hình thức trong hàm

Như ta đã biết, hàm chỉ trả về một giá trị biến, nên muốn trả về nhiều biến ta phải sử dụng thêm biến toàn cục trong hàm. Điều này làm thay đổi giá trị biến toàn cục sau khi gọi hàm, sẽ gây khó kiểm soát giá trị của biến toàn cục trong một số trường hợp. Sử dụng con trỏ làm thông số hình thức trong hàm sẽ tăng thêm các giá trị trả về và tránh phải khai báo nhiều biến.

Ví dụ 5.32: Viết một hàm hoán vị giá trị 2 biến.

```
#include <avr/io.h>
```

```

int a,b;
void swap2(int *p1,int *p2)
{
    int tam;
    tam=*p1;//tam=(p1)
    *p1=*p2;//(p1)=(p2)
    *p2=tam;//(p2)=tam
}
int main()
{
    a=0x1234;
    b=0x5678;
    swap2(&a,&b);
    while(1);
}

```

5.7 Mảng(Arrays)

Mảng là kiểu dữ liệu đặc biệt bao gồm tập hợp các biến hay hằng có cùng kiểu dữ liệu, được sử dụng rất hiệu quả trong việc truy xuất nguyên một vùng nhớ. Việc truy xuất mảng cũng tương tự như dùng phương pháp định vị gián tiếp có chỉ số độ lệch.

❖ Cú pháp:

type tên_mảng[n];

Hoặc:

type tên_mảng[n]={v0,v1,v2,...,vn};

Hoặc:

type tên_mảng[]={v0,v1,v2,...,vn};

- **type:** kiểu dữ liệu của các biến trong mảng

- **tên_mảng:** tương tự tên biến

- **n:** chỉ số lượng biến được định nghĩa trong mảng, không giới hạn về lý thuyết (tùy thuộc dung lượng bộ nhớ)

- **[]:** ký hiệu là biến dạng mảng

- **v0,v1,v2,...,vn:** n biến có cùng kiểu dữ liệu **type** trong mảng

- **{ }:** ký hiệu bao gồm các phần tử trong mảng

Một mảng tối thiểu phải có một phần tử.

Ví dụ: **char str1[5];**

char str2[]={1,2,4,9,16,25};

Mảng được khai báo ở trên gọi là mảng 1 chiều. Trường hợp muốn khai báo một chuỗi ký tự ta có thể sử dụng mảng với chuỗi ký tự đặt trong dấu (" "), ví dụ như sau:

char str[]="Xin chao! ";

Tương đương như:

char str[]={0x58,0x69,0x6e,0x61,0x6f,0x20};

5.7.1 Truy xuất mảng

Có 2 cách truy xuất mảng gồm truy xuất trực tiếp và truy xuất bằng con trỏ.

1. Truy xuất trực tiếp mảng

Để truy xuất một phần tử (biến) của mảng, chỉ cần gọi tên mảng và chỉ số phần tử, chính là vị trí thứ tự của phần tử nằm trong mảng. Quy định chỉ số bắt đầu mảng luôn luôn bằng 0 ứng với phần tử đầu tiên trong mảng.

Ví dụ: **char str1[5];**

char str2[]={1,2,4,9,16,25};

char x;

x=str2[4];//x=16

str1[1]=x;//str1[5]={x,16,x,x,x}, biến thứ 2 mảng str1=16

Hoặc có thể viết:

str1[1]=str2[4];

2. Truy xuất mảng bằng con trỏ

Sử dụng con trỏ chỉ địa chỉ của phần tử của mảng:

Tiếp theo ví dụ trên:

```
char *p;  
p=&str2[3];//p trỏ địa chỉ biến thứ 3 của mảng str2  
x=*p;//x=9
```

Hoặc có thể viết:

```
p=&str2;//p trỏ địa chỉ đầu mảng ứng với biến str2[0]  
x=(p+3);//p+3 trỏ đến biến str2[3],x=9
```

- Trường hợp str2 là biến mảng,có thể viết phát biểu p trỏ vào địa chỉ đầu mảng bỏ dấu &: **p=str2**

Ví dụ 5.33: Viết một chương trình nhập từ PortA số nhị phân từ 00-0f,xuất giá trị bình phương của số nhập ra PortB và lưu nó vào SRAM địa chỉ cụ thể tùy chọn.

Giải:

Ta sử dụng mảng làm bảng tra giá trị bình phương của số nhập từ PortA che lấy 4 bit thấp,số nhập chính là chỉ số của giá trị bình phương của nó.Ta chọn địa chỉ ô nhớ SRAM=0x115 lưu giá trị bình phương.Việc chọn địa chỉ ô nhớ này sẽ được trình bày ở phần sau.

```
#include <avr/io.h>  
char a,*p;  
char sqr[]={0,1,4,9,16,32,36,49,64,81,100,121,144,169,196,225};  
const int S_buf=0x115;  
int main()  
{  
    DDRA=0x00;//PortA input  
    PORTA=0xff;//R kéo lên PortA  
    DDRB=0xff;//PortB output  
    PORTB=0x00;//xóa PortB=0x00  
    while(1)  
    {  
        a=PINA&0x0f;//nhập số từ PortA che 4 bit thấp  
        a=sqr[a]; //tra bảng lấy bình phương a gán vào a  
        p=S_buf; //p trỏ địa chỉ ô nhớ SRAM  
        PORTB=a; //xuất a ra PortB  
        *p=a; //cắt a vào ô nhớ địa chỉ p  
    }  
}
```

Ta cũng có thể viết khối phát biểu trong while(1) chỉ cần sử dụng con trỏ truy xuất mảng như sau:

```
...  
while(1)  
{  
    p=sqr; //p trỏ địa chỉ đầu bảng tra  
    PORTB=*(p+(PINA&0x0f));//cộng p với số nhập từ PortA sau khi che 4 bit thấp,  
                           //xuất bình phương số nhập ra PortB  
    p=S_buf; //p trỏ địa chỉ đầu bảng tra  
    *p=PORTB; //cắt số bình phương vào ô nhớ  
}
```

- Để chọn địa chỉ ô nhớ SRAM lưu giá trị bình phương,ta cần lưu ý trình tự lưu trữ các biến và hằng vào vùng nhớ SRAM khi biên dịch và chạy chương trình trên như sau:
- Lưu 16 biến của mảng sqr từ địa chỉ 0x100 - 0x10f
 - Lưu biến a(1 byte),biến con trỏ p(2 byte)vào 3 địa chỉ tiếp theo 0x110-0x112
 - Như vậy S_buf tối thiểu phải từ 0x113 trở lên mới không bị đè lên các biến trên!

5.7.2 Khai báo và truy xuất mảng hàng trong Flash ROM(không gian “.text”)

Như đã trình bày trong mục 5.2.4,trong ví dụ trên do bản thân khai báo mảng sqr trong bộ nhớ không gian “.data”(SRAM)mặc định, mảng sqr cũng có mặt trong vùng nhớ chương trình hay không gian “.text”(Flash ROM) khi biên dịch.Ta có thể khai báo mảng sqr đặt trong Flash ROM để tiết kiệm không gian SRAM.

Để truy xuất phần tử dữ liệu trong mảng đặt trong Flash ROM,ta sử dụng macro trong thư viện (pgmspace.h) cung cấp:

❖ Cú pháp: **pgm_read_byte(addr16)**

- **addr16**: địa chỉ phần tử trong mảng 16 bit

Ta viết lại ví dụ 5.33 sử dụng khai báo và truy xuất mảng trong Flash ROM.

Ví dụ 5.34:

```
#include <avr/io.h>
#include <avr/pgmspace.h> //thư viện sử dụng các hàm macro trong flash
char *p;
const char sqr[] PROGMEM={0,1,4,9,16,32,36,49,64,81,100,121,144,169,196,225};
//khai báo mảng trong flash
const int S_buf=0x115;
int main()
{
    DDRA=0x00;//PortA input
    PORTA=0xff;//R kéo lên PortA
    DDRB=0xff ;//PortB output
    PORTB=0x00;//xóa PortB=0x00
    while(1)
    {
        p=&sqr; //p trả địa chỉ đầu bảng tra
        PORTB=pgm_read_byte(p+(PIN_A&0x0f));//cộng p với số nhập từ PortA sau khi che 4
        //bit thấp,tra địa chỉ flash xuất bình phương số nhập ra PortB
        p=S_buf; //p trả địa chỉ đầu bảng tra
        *p=PORTB; //cắt số bình phương vào ô nhớ
    }
}
```

5.7.3 Sử dụng mảng trong hàm

1. Thông số hình thức

Ta có thể khai báo mảng làm thông số hình thức trong hàm,có 3 cách:

- Trực tiếp biến mảng kích thước cụ thể,ví dụ int funct_h(char arr[10],char x)
- Trực tiếp biến mảng không có kích thước,ví dụ int funct_h(char arr[],char x)
- Gián tiếp qua biến con trả chỉ địa chỉ mảng,ví dụ int funct_h(char *p,char x)

2. Giá trị trả về từ hàm

Như ta đã biết hàm chỉ trả về duy nhất giá trị 1 biến,do đó không thể trả về giá trị của cả mảng! Để có thể trả về giá trị cả mảng,ta phải dùng định vị gián tiếp bằng cách trả về con trả địa chỉ của mảng, sau đó từ con trả địa chỉ mới truy xuất được mảng.

Khai báo hàm trả về địa chỉ con trả của biến:

void * tên_hàm(danh sách thông số)

Biến trong hàm thuộc loại biến cục bộ nên để giữ nguyên giá trị biến khi trả về ta phải khai báo **static** cho biến:

static type tên_bien;

Ta xem ví dụ minh họa sau đây,trả về biến mảng từ gọi hàm.

Ví dụ 5.35:

```
#include <avr/io.h>
char a,*p;
char str[]={0,1,2,3,4,5,6,7,8,9};
void *fp(char arr[],char n)//định nghĩa hàm trả về con trả địa chỉ mảng
```

```

{
    char j;
    static char rn[10];//khai báo biến mảng rn static
    for(j=0;j<=9;j++) //lặp vòng 10 lần
        rn[j]=n*arr[j];//nhân mảng arr[] cho n
    return rn;//trả về biến mảng
}
int main()
{
    while (1)
    {
        p=fp(str,2)+2;//gọi hàm tính 2xstr[] trả về con trỏ địa chỉ mảng p,cộng p thêm 2
        a=*p;//truy xuất phần tử địa chỉ p của mảng trả về từ hàm(phần tử thứ 4)
    }
}

```

Trong ví dụ trên giá trị phần tử thứ 4 của mảng trả về từ hàm a=4=2x2.

5.7.4 Mảng nhiều chiều

Ta có thể khai báo mảng nhiều chiều, ngôn ngữ C xem khai báo mảng nhiều chiều như khai báo mảng trong mảng. Một ứng dụng tiêu biểu của mảng 2 chiều là bảng tra hàng/cột hay ma trận.

Ví dụ khai báo ma trận 4x4 như sau:

```

char matrix[][]={{0,1,2,3,
                  4,5,6,7
                  8,9,A,B
                  C,D,E,F
                  }};

```

Để truy xuất phần tử a_{ij} trong ma trận, ta chỉ cần gán $matrix[i][j]$ với $i,j=0-3$.

Ví dụ:

```

char a;
a=matrix[2][3];//a=B

```

Ví dụ 5.36: Trên bàn phím điện thoại bố trí các phím như sau:

h0	1	2	3
h1	4	5	6
h2	7	8	9
h3	*	0	#
c0		c1	c2

Giả sử hàm `key_code()` đọc giá trị từ PortA trả về giá trị `char` thể hiện vị trí phím nhấn tương ứng 4 bit cao minh họa vị trí hàng và 4 bit thấp minh họa vị trí cột. Nếu có 1 phím nhấn, bit hàng và cột tương ứng sẽ mức 0, các bit còn lại mức 1. Ví dụ phím 8 nhấn tương ứng vị trí hàng 2 cột 1, hàm `key_code()` sẽ trả về giá trị 10111101. Trường hợp không có phím nhấn giá trị đọc từ PortA bằng 0x00. Viết một đoạn chương trình trả về giá trị mã ASCII phím nhấn nếu có phím nhấn, và trả về giá trị 0xff nếu không có phím nhấn.

Giai:

```

#include <avr/io.h>
char a;
const char key(chr[4][3]= //khai báo mảng 2 chiều giá trị phím=mã ASCII
{ '1','2','3',
  '4','5','6',
  '7','8','9',
  '*', '0', '#'
};
char key_code() //hàm đọc mã vị trí phím
{

```

```

char x;
x=PINA;
return x;
}
char bit_pos(char b,const char n) //hàm nhận dạng hàng,cột phím nhấn
{
    char i,tam;
    tam=0xfe;          //khởi động mã quét
    for(i=0;i<=n;i++)//lặp vòng n +1 lần tối đa
    {
        if(b==tam)  //tim vị trí bit=0
        {
            b=i;      //gán vị trí bit
            break;    //thoát
        }
        else
            tam=(tam<<1)+1;//dịch sang bit kế tiếp
    }
    return b;//trả về vị trí bit=0
}
char key_val(char y) //hàm xác định giá trị phím
{
    char h,c;
    h=(y>>4)|0xf0; //dịch mã hàng xuống 4 bit thấp,đặt 4 bit cao=1
    h=bit_pos(h,3); //gọi hàm trả về vị trí hàng phím nhấn(=0)
    c=y|0xf0;        //đặt 4 bit cao mã cột =1
    c=bit_pos(c,2); //gọi hàm trả về vị trí cột phím nhấn(=0)
    return key_chr[h][c];//trả về giá trị ASCII phím nhấn
}
int main()
{
    DDRA=0x00;
    PORTA=0xff;
    while(1)
    {
        a=key_code(); //đọc mã vị trí phím
        if(a!=0)       //có phím nhấn?
            a=key_val(a);//có,gọi hàm xác định giá trị phím
        else
            a=0xff;    //không,trả về mã 0xff
    }
}

```

❖ Câu hỏi ôn tập

- Để lưu giá trị một biến vào địa chỉ SRAM=0x200 phải viết như thế nào?
- Viết lại hàm chuyển đổi từ số nhị phân 8 bit sang BCD không nén 3 digit,cắt trong 3 ô nhớ SRAM liên tiếp có địa chỉ đầu cho trước.Ngõ vào tham số của hàm là số nhị phân và địa chỉ đầu SRAM.
- Viết một hàm trả về 2 lũy thừa của biến nhập từ 0-15,sử dụng phương pháp tra bảng.
- Viết một hàm xuất chuỗi ký tự tối đa 16 ký tự,có ký tự kết thúc là Null=\$00 ra Port output(đã định nghĩa),thoát khỏi hàm khi gặp ký tự Null hoặc hết 16 ký tự tùy điều kiện nào đến trước.
- Khai báo chuỗi ký tự “Nhiệt do (°C): “ với mã ASCII của dấu (°)=\$fd.Viết chương trình xuất chuỗi ký tự trên ra PortB,áp dụng hàm ở câu 4,lập vòng 20 lần.
- Viết một hàm nhập 1 chuỗi dữ liệu(byte) từ Port đã định nghĩa trước,kết thúc chuỗi là mã CR=\$0d, cắt vào vùng nhớ SRAM có địa chỉ đầu cho trước.

5.8 Cấu trúc(Structures)

Cấu trúc là kiểu dữ liệu đặc biệt gồm tập hợp nhiều biến có các kiểu dữ liệu khác nhau, để biểu diễn các đặc điểm, thông số của một đối tượng. Ví dụ một hồ sơ khám sức khỏe tóm tắt của một người gồm:

- Họ và tên
- Mã định danh
- Ngày sinh
- Chiều cao
- Cân nặng

Các thông tin trên của một người trong hồ sơ có thể có các kiểu dữ liệu khác nhau, sẽ được gán chung vào một biến thuộc kiểu dữ liệu cấu trúc để dễ truy xuất. Một số ngôn ngữ cấp cao thường gọi kiểu cấu trúc là bản ghi(records).

Để sử dụng biến kiểu cấu trúc, trước tiên ta phải khai báo kiểu cấu trúc.

❖ Cú pháp:

```
struct [tên_cấu_trúc]
{
    type thành_viện_1;
    type thành_viện_2;
    ...
    type thành_viện_n;
}[biến_1,biến_2,...];
```

- tên_cấu_trúc: tương tự như tên biến, có thể bỏ qua

- type thành_viện_i(i=1-n) : kiểu dữ liệu và tên thành viên (biến) trong cấu trúc

- biến_1, biến_2 : danh sách các biến thuộc kiểu cấu trúc, có thể bỏ qua

Để phân biệt biến có kiểu dữ liệu riêng trong khai báo kiểu cấu trúc, ta gọi là thành viên trong cấu trúc.

Ví dụ 5.37: Khai báo biến kiểu cấu trúc hồ sơ khám sức khỏe một nhân sự.

```
struct health_prf
{
    char name[20];           // họ và tên kiểu array
    long id_p;                // mã định danh kiểu long
    char day_bth;             // ngày sinh kiểu char
    char month_bth;           // tháng sinh kiểu char
    int year_bth;              // năm sinh kiểu int
    char height;               // chiều cao kiểu char
    float weight;                // cân nặng kiểu float
}; id_var1,id_var2;
```

Trong khai báo trên, biến id_var1, id_var2 là các biến kiểu cấu trúc health_prf có các thành viên có kiểu dữ liệu khai báo trong dấu ({}).

Trong trường hợp khai báo biến kiểu cấu trúc sau khi khai báo kiểu cấu trúc health_prf, ta khai báo như sau:

```
struct health_prf id_var3,id_var4;
```

Các thành viên trong cấu trúc có thể có kiểu dữ liệu một cấu trúc đã khai báo. Trong ví dụ trên ta khai báo riêng cấu trúc ngày sinh như sau:

```
struct dob
{
    char day_bth;           // ngày sinh kiểu char
    char month_bth;           // tháng sinh kiểu char
    int year_bth;              // năm sinh kiểu int
};
```

Khai báo lại cấu trúc health_prf như sau:

```
struct health_prf
{
    char name[20];           // họ và tên kiểu array
    long id_p;                // mã định danh kiểu long
    struct dob date;           // ngày sinh kiểu struct
```

```

char height;           //chiều cao kiểu char
float weight;          //cân nặng kiểu float
}id_var1,id_var2;

```

5.8.1 Truy xuất các thành viên trong cấu trúc

Để truy xuất bất kỳ một thành viên nào trong cấu trúc đã khai báo, ta gọi tên biến cấu trúc, toán tử(.) và sau là tên thành viên: **biến_cấu_trúc.tên_thành_viện**

Ví dụ 5.38: Gán các giá trị cho các thành viên của biến cấu trúc id_var1,id_var2 trong cấu trúc health_prf đã khai báo ở ví dụ 5.37.

Giải:

```

.....
//khai báo các biến
int a;
char i,c,b[20];
float d;

.....
//gán các giá trị cho các thành viên biến id_var1
strcpy(id_var1.name,"Nguyen Van An");//id_var1.name[20] = "Nguyen Van An"
id_var1.id_p=12345678;
id_var1.day_bth=12;
id_var1.month_bth=6;
id_var1.year_bth=1999;
id_var1.height=169;
id_var1.weight=65.5;
//gán các giá trị cho các thành viên biến id_var2
strcpy(id_var2.name,"Trinh Van Binh");//id_var2.name[20] = "Trinh Van Binh"
id_var2.id_p=12347890;
id_var2.day_bth=25;
id_var2.month_bth=10;
id_var2.year_bth=2001;
id_var2.height=172;
id_var2.weight=61.7;
//-----
//đọc các thành viên cấu trúc
//-----
a=id_var1.height;//a=169
for(i=0;i<=19;i++)
    b[i]=id_var1.name[i];//b="Nguyen Van An"
c=id_var2.day_bth;//c=25
d=id_var2.weight;//d=61.7

```

Trường hợp khai báo biến date kiểu struct dob, ta gán cho biến id_var1 như sau:

```

id_var1.date.day_bth=12;
id_var1.date.month_bth=6;
id_var1.date.year_bth=1999;

```

Tương tự như trên cho biến id_var2.

- Trong ví dụ 5.38 ta có sử dụng hàm thư viện strcpy dùng để chép chuỗi ký tự cho mảng 1 chiều. Hàm strcpy tự động chép thêm cho mảng ký tự kết thúc ký hiệu Null=0x00 ở cuối chuỗi ký tự. Tất nhiên chiều dài của mảng phải không nhỏ hơn tổng số ký tự trong chuỗi cộng thêm ký tự Null.

❖ **Cú pháp hàm strcpy:** **strcpy(tên_mảng,"chuỗi_ký_tự")**

5.8.2 Cấu trúc làm thông số hình thức của hàm

Ta sử dụng biến kiểu cấu trúc làm thông số hình thức của hàm tương tự như các loại biến kiểu khác.

Ví dụ 5.39: Tiếp theo ví dụ 5.37 và 5.38, ta thay đoạn đọc các thành viên cấu trúc health_prf bằng hàm.

```

.....
//khai báo các biến toàn cục
long y;
char i,x[20];
float d;
.....
void get_par(struct health_prf var) //biến cấu trúc var làm thông số hình thức
{
    for(i=0;i<=19;i++)
        x[i]=var.name[i];//đọc thành viên name
    y=var.id_p;//đọc thành viên mã id
}
.....
//đọc họ và tên.mã id của biến cấu trúc
get_par(id_var1);

```

Kết quả trả về(dạng mã Hex)

```

x=(4e 67 75 79 65 6e 20 56 61 6e 20 41 6e 00 00 00 00 00 00 00)=”Nguyen Van An”
y=0x00bc614e=12345678
    Null

```

5.8.3 Con trỏ cấu trúc

Ta cũng có thể sử dụng con trỏ truy xuất cấu trúc theo định dạng sau:

- Khai báo con trỏ cấu trúc :

```
struct tên_cấu_trúc * tên_con_trỏ_cấu_trúc;
```

Để truy xuất cấu trúc:

- Gán con trỏ địa chỉ biến cấu trúc:

```
con_trỏ_cấu_trúc=&biến_cấu_trúc;
```

- Sử dụng toán tử -> truy xuất thành viên của biến cấu trúc:

```
biến=con_trỏ_cấu_trúc -> thành_viện;
```

Ví dụ 5.40: Tiếp theo ví dụ từ 5.37 đến 5.39, ta viết thêm hàm truy xuất chiều cao và cân nặng:

```

.....
//khai báo các biến toàn cục
char h;
float w;
struct health_prf *p;//biến con trỏ cấu trúc
.....
void get_par2(struct health_prf *var_hw) //con trỏ cấu trúc làm thông số hình thức
{
    h=var_hw->height;//đọc thành viên height qua biến con trỏ cấu trúc
    w=var_hw->weight;//đọc thành viên weight qua biến con trỏ cấu trúc
}
.....
p=&id_var1;//gán địa chỉ biến cấu trúc
get_par2(p);//đọc các chỉ số chiều cao, cân nặng

```

5.9 Hợp nhất(Unions)

Hợp nhất cũng là kiểu dữ liệu đặc biệt có định dạng khai báo hoàn toàn giống như kiểu cấu trúc.

❖ Cú pháp

```
union [tên_hợp_nhất]
{
    type thành_viện_1;
    type thành_viện_2;
    ...
    type thành_viện_n;
}[biến_1,biến_2,...];
```

Điểm khác biệt giữa hợp nhất và cấu trúc ở vấn đề cấp phát bộ nhớ. Trong kiểu cấu trúc, trình biên dịch sẽ dự trữ riêng từng vùng nhớ có độ dài tương ứng với khai báo kiểu dữ liệu của từng thành viên. Với một biến cấu trúc, trình biên dịch sẽ dự trữ vùng nhớ bằng tổng số byte dự trữ cho từng thành viên trong cấu trúc. Đối với kiểu hợp nhất, các thành viên trong hợp nhất mặc dù có kiểu dữ liệu khác nhau, nhưng đều sử dụng chung một vùng nhớ có độ dài chính xác bằng độ dài vùng nhớ dài nhất của thành viên trong hợp nhất. Ta xem lại ví dụ 5.37 để so sánh sự khác biệt về cấp phát vùng nhớ giữa hai kiểu dữ liệu này.

Nếu khai báo kiểu cấu trúc:

```
struct health_prf
{
    char name[20];           // 20 byte
    long id_p;               // 4 byte
    char day_bth;            // 1 byte
    char month_bth;          // 1 byte
    int year_bth;            // 2 byte
    char height;              // 1 byte
    float weight;             // 4 byte
}id_var1,id_var2;
```

Mỗi biến kiểu cấu trúc sẽ có dự trữ 33 byte ô nhớ.

Nếu khai báo kiểu hợp nhất:

```
union health_prf
{
    char name[20];           // 20 byte
    long id_p;               // 4 byte
    char day_bth;            // 1 byte
    char month_bth;          // 1 byte
    int year_bth;            // 2 byte
    char height;              // 1 byte
    float weight;             // 4 byte
}id_var1,id_var2;
```

Mỗi biến kiểu hợp nhất chỉ được dự trữ 20 byte ô nhớ cho toàn bộ các thành viên.

Nếu thực hiện đoạn chương trình sau đối với biến kiểu hợp nhất:

```
//gán các giá trị cho các thành viên biến id_var1
strcpy(id_var1.name,"Nguyen Van An");//id_var1.name[20] = "Nguyen Van An"
id_var1.id_p=12345678;
id_var1.day_bth=12;
id_var1.month_bth=6;
id_var1.year_bth=1999;
id_var1.height=169;
id_var1.weight=65.5;
```

Kết quả đọc được trong vùng nhớ dự trữ cho biến id_var1 là:

00 00 83 42 65 6e 20 56 61 6e 20 41 6e 00 00 00 00 00 00

Bốn byte thấp biểu diễn ("Nguy" = 4e 67 75 79) sau cùng đã bị gán bởi giá trị cân nặng (id_var1.weight = 65.5 = 00 00 83 42).

Do đó, để tránh mất dữ liệu, ta chỉ nên truy xuất một thành viên của biến kiểu hợp nhất tại một thời điểm. Kiểu hợp nhất được sử dụng để cấp phát bộ nhớ cho biến hiệu quả và tiết kiệm bộ nhớ.

Ví dụ sau đây minh họa áp dụng kiểu hợp nhất.

Ví dụ 5.41: Viết một đoạn chương trình nhập một từ 16 bit, byte cao từ PortB, byte thấp từ PortA.

Giải:

Trước tiên ta thực hiện cách thông thường:

```
#include <avr/io.h>
int io_var;
int main(void)
{
```

```

DDRA=0x00;
PORTA=0xff;
DDRB=0x00;
PORTB=0xff;
while(1)
{
    io_var=PINB<<8;
    io_var |=PINA;
    ...
}

```

Cách sau đây áp dụng kiểu union:

```

#include <avr/io.h>
union io_16          //khai báo union dài 2 byte
{
    char in_port[2];
    int port_w;
};
int io_var;
int import_16(char io_1,char io_h) //hàm đọc data từ Port
{
    union io_16 uf;
    uf.in_port[0]=io_1; //đọc data từ Port byte thấp
    uf.in_port[1]=io_h;//đọc data từ Port byte cao
    return uf.port_w; //trả về thành viên port_w kiểu int
}
int main(void)
{
    DDRA=0x00; //khai báo các Port input và R kéo lên
    PORTA=0xff;
    DDRB=0x00;
    PORTB=0xff;
    while(1)
    {
        io_var=import_16(PINA,PINB); //đọc data từ PortB:PortA
        ...
    }
}

```

5.10 Trường bit(Bitfields)

Trường bit là một kiểu dữ liệu đặc biệt làm việc trên bit, được gán cho các thành viên trong một cấu trúc hoặc hợp nhất. Áp dụng trường bit rất thuận tiện cho những phát biểu chuyên xử lý bit và tiết kiệm được bộ nhớ.

❖ Cú pháp:

type tên_thành_viện_b : n;

- type: các kiểu dữ liệu số thông thường là char,unsigned char,int,unsigned int

- tên_thành_viện_b: tên thành viên trong kiểu cấu trúc hoặc hợp nhất

- n : độ dài tính bằng bit biểu diễn tên thành viên b, không được lớn hơn độ dài bit kiểu type

Ví dụ để biểu diễn trạng thái của 4 tác vụ theo dạng TRUE=1 hay FALSE=0, ta khai báo 4 biến trạng thái như sau:

```

char dir_flg;
char sw_flg;
char stop_flg;
char run_flg;

```

```
...
dir_flg=0x01;//TRUE=1
sw_flg=0x00;//FALSE=0
```

Thật ra mỗi biến trạng thái chỉ cần 1 bit biểu diễn,tổng cộng cần 4 bit,nhưng phải sử dụng đến 4 byte bằng 32 bit bộ nhớ!

Bây giờ nếu ta khai báo theo kiểu cấu trúc&trường bit như sau,sẽ chỉ tốn 1 byte chứa 4 bit trạng thái cho biến cấu trúc state:

```
struct reg_flag
{
    char dir_flg : 1;
    char sw_flg : 1;
    char stop_flg: 1;
    char run_flg: 1;
}state;
```

Truy xuất các thành viên trong cấu trúc trên như truy xuất bit:

```
state.dir_flg=0;
state.sw_flg=1;
```

Ta xem ví dụ sau áp dụng cấu trúc trường bit trong các tác vụ xử lý bit.

Ví dụ 5.42:

```
#include <avr/io.h>
struct reg_flag      //khai báo cấu trúc có trường bit
{
    char dir_flg : 1;
    char sw_flg : 1;
    char stop_flg: 1;
    char run_flg: 1;
}state;

int main(void)
{
    DDRA=0x00;      //PortA input
    PORTA=0xff;     //R kéo lên PortA
    DDRB=0xf0;      //PB3:PB0 input,PB7:PB4 output
    PORTB=0x0f;     //R kéo lên PB3:PB0,PB7:PB4=0
    DDRC=0xff;      //PortC output
    PORTC=0x00;     //PortC=0x00
    while (1)
    {
        //gán các giá trị bit theo trạng thái đọc từ PortB
        if((PINB&(1<<PB0))==0)
            state.dir_flg=1;
        else
            state.dir_flg=0;
        if((PINB&(1<<PB1))==0)
            state.sw_flg=1;
        else
            state.sw_flg=0;
        if((PINB&(1<<PB2))==0)
            state.stop_flg=1;
        else
            state.stop_flg=0;
        if((PINB&(1<<PB3))==0)
            state.run_flg=1;
        else
            state.run_flg=0;
```

```

if(state.dir_flg==0) //xét trạng thái bit dir_flg
    PORTC=PIN; //thực hiện khi dir_flg=0
else
    PORTC=~PIN; //thực hiện khi dir_flg=1
if(state.sw_flg==0) //xét trạng thái bit sw_flg
    PORTB =0xaf; //PB7:PB4=1010 khi sw_flg=0
if(state.stop_flg==0) //xét trạng thái bit stop_flg
    PORTB |=0xf0; //PB7:PB4=1111 khi stop_flg=0
else if(state.run_flg==0)//stop_flg=1,xét trạng thái bit run_flg
    PORTB &=0x0f; //PB7:PB4=0000 khi run_flg=0
}
}

```

5.11 Toán tử **typedef**

Toán tử **typedef** cho phép định nghĩa lại tên kiểu dữ liệu mới, mục đích để người đọc chương trình hiểu và liên hệ được một số đặc điểm, chức năng của biến.

Ví dụ:

```

typedef unsigned char byte; //định nghĩa byte dài 1 byte
typedef unsigned char word; //định nghĩa word dài 2 byte
byte x;//biến độ dài byte
word y;//biến độ dài 2 byte

```

Toán tử **typedef** tương tự như **#define**, tuy nhiên **typedef** được sử dụng trong các khai báo và biến dịch trực tiếp, trong khi **#define** là một tiền xử lý được thực hiện trước khi biên dịch (xem mục sau).

❖ Câu hỏi ôn tập

1. Khai báo một kiểu biến cấu trúc chỉ các thông tin:

- Tên kênh đo 15 ký tự
- Giá trị điện áp từ 0-255(V)
- Giá trị dòng điện từ 0.00 -9999(mA)

2. Viết các phát biểu ghi các giá trị vào biến cấu trúc:

- “Kênh 01”(01 là mã kênh)
- 220
- 1234

3. Đọc các giá trị biến cấu trúc ở câu 2 cất vào vùng nhớ SRAM địa chỉ đầu 0x200.

4. Giả sử từ địa chỉ 0x200 vùng nhớ SRAM có lưu các giá trị của các biến cấu trúc như câu 2 và 3.

Dựa vào đó viết một hàm đọc các giá trị của biến cấu trúc nếu nhập vào mã kênh (quy định mã kênh 2 chữ số từ 00 - 99 nằm trong chuỗi ký tự tên kênh đo)

5. Tại một thời điểm chỉ có thể 1 chân ngõ vào PortD mức 0. Viết một hàm trả về biến 8 bit định dạng như sau :

- Nếu có 1 chân ngõ vào PortD mức 0, bit0=0, ba bit tiếp theo biểu diễn trọng số chân port
- Nếu không có chân nào của PortD mức 0, bit0=1

5.12 Tiền xử lý(Preprocessors) và macro

Các tiền xử lý trong C(CPP- C PreProcessors) không phải là một thành phần của trình biên dịch, mà là một bước riêng biệt trong quá trình biên dịch. CPP chỉ là một công cụ thay thế văn bản ra lệnh cho trình biên dịch thực hiện quá trình chuẩn bị trước khi biên dịch thật sự.

Các phát biểu CPP đều có dấu (#) đứng đầu và theo liền sau không có khoảng cách là tên chỉ dẫn, và phải được đặt ở đầu dòng.

5.12.1 Các CPP thông dụng

Sau đây là bảng tóm tắt các CPP thông dụng.

STT	CPP& Chỉ dẫn	Mô tả
1	#define	Thay thế một macro CPP (định nghĩa 1 danh hiệu)
2	#include	Gắn 1 file.h cũ thể từ ngoài chương trình
3	#undef	Xóa định nghĩa 1 macro CPP
4	#ifdef	Trả về TRUE nếu macro này đã được định nghĩa

5	#ifndef	Trả về TRUE nếu macro này chưa được định nghĩa
6	#if	Kiểm tra điều kiện thỏa mãn tại thời điểm biên dịch
7	#else	điều kiện #if không thỏa mãn
8	#elif	#else và #if cùng 1 phát biểu
9	#endif	Kết thúc điều kiện CPP
10	#error	In thông điệp báo lỗi trên stderr
11	#pragma	Thông báo các yêu cầu đặc biệt(đã được chuẩn hóa)

Ví dụ 5.43: Các dòng phát biểu sau đây minh họa việc sử dụng các CPP.

```
#include <avr/io.h> //gắn hàm thư viện định nghĩa avr-io vào trong chương trình
#define Num 10      //định nghĩa Num=10
#define Import PINB //định nghĩa Import=PINB
//-----
unsigned char a,x,y,z[10];
int main()
{
    x=Num;      //x=10
    y=Import;   //y=PINB
#
# undef Num      //xóa định nghĩa Num trước đó
# define Num 20   //định nghĩa lại Num=20
//
#ifndef MSG           //nếu MSG chưa được định nghĩa
    #define MSG "Xin chào!"//định nghĩa MSG="Xin chào!"
    strcpy(z,MSG);        //chép chuỗi ký tự MSG vào mảng z
#endif             //kết thúc CPP #ifndef
//
#ifndef Import         //nếu đã định nghĩa Import
    #define Outport PORTC //định nghĩa Outport=PortC
#else                //nếu chưa định nghĩa Import
    #define Outport PORTB //định nghĩa Outport=PortB
#endif
    Outport=Import;     //PortC=PinB
    x+=Num;            //x=10+20=30
    a=z[0];            //a='X'=0x58
}
//-----
```

5.12.2 Macro

Macro trong C tương tự như macro trong hợp ngữ, là tập hợp một đoạn phát biểu được định nghĩa bằng một danh hiệu gọi là tên macro. Macro trong C được xem như là một chỉ dẫn CPP do người lập trình tự biên soạn, nên về hình thức trình bày tương tự như trình bày CPP. Khi biên dịch chương trình, tại vị trí có tên macro sẽ được thay đổi bằng đoạn phát biểu đã định nghĩa trước đó.

Định dạng một macro như sau:

- Phải có CPP #define bắt đầu
- Các giá trị hằng, biến, thông số hình thức(nếu có) của macro không cần khai báo kiểu dữ liệu
- Thân macro phải được viết cùng một dòng với tên macro và CPP
- Kết thúc macro không có dấu (;) và xuống dòng mới

❖ Cú pháp:

#define Tên_macro [(thân macro)]

Có thể phân loại macro ra làm 3 dạng: dạng đối tượng, dạng hàm, dạng chuỗi

1. Macro dạng đối tượng(Object-Like Macros)

Macro dạng đối tượng có thân macro là hằng số hoặc biến

Ví dụ: #define Num 20

#define Import PINB

#define Pi 3.14159

```
#define MSG "Xin chào!"
```

2. Macro dạng hàm(Function-Like Macros)

Macro dạng hàm tương tự như hàm,có thể có thông số hình thức,thân macro biểu diễn phát biểu là biểu thức có chứa các thông số hình thức.

Ví dụ 5.44: Viết macro cộng 2 giá trị.

Giải:

```
#include <avr/io.h>
#define add(a,b) (a+b) //macro cộng 2 biến
int x,y,z;
int main(void)
{
    while (1)
    {
        x=100;
        y=200;
        z=add(x,y);
    }
}
```

Trong định nghĩa macro dạng hàm,2 thông số hình thức a,b không cần khai báo kiểu dữ liệu.Chỉ đến khi sử dụng trong chương trình ta mới định nghĩa giá trị thay thế vào kiểu int.Đây chính là ưu điểm rất linh động và thuận lợi của macro!

3. Macro dạng chuỗi(Chain-Like Macros)

Khi một macro được định nghĩa bao gồm một hoặc nhiều macro đã được định nghĩa khác,ta gọi macro có dạng chuỗi.

Ví dụ 5.45: Viết một macro tính diện tích hình tròn.

Giải:

```
#include <avr/io.h>
#define Pi 3.14159
#define Cir_Area(x) ((x*x)*Pi)//macro tính diện tích hình tròn dạng chuỗi
float r,S;
int main(void)
{
    while (1)
    {
        r=0.5;
        S=Cir_Area(r);
    }
}
```

Kết quả cho S=0x3f490fd0=0.788330

4. Các toán tử trong macro

- **Toán tử nối dòng (\):** dùng để báo nối dòng khi thân macro kéo dài quá 1 dòng.Do định dạng macro xuống dòng là kết thúc định nghĩa macro,nên khi trình bày thân macro kéo dài quá 1 dòng,trước khi xuống dòng ta thêm dấu () báo trình biên dịch biết nội dung dòng dưới cùng dòng với dòng trên.

Ví dụ định nghĩa macro tìm số lớn hơn trong 2 số:

```
#define Max(a,b) ((a > b) ? a : b)
viết thành 2 dòng sử dụng dấu nối dòng:
#define Max(a,b) ((a > b) \
? a : b)
```

- **Toán tử xâu chuỗi(#)(stringize operator):** sử dụng để chuyển thông số hình thức thành chuỗi ký tự khi biên dịch.Dấu (#) thay cho dấu (" ")

Ví dụ:

```
#define chr num(a) (#a)
#define MSG(st) (#st)
```

```

...
char j[2],arr[10];
strcpy(j,chr_num(1));
strcpy(arr,MSG(Xin chao!));

```

Kết quả trả về mã ASCII chuỗi ký tự,mã Null=00 thêm vào báo kết thúc chuỗi:

```

j={31,00}
arr={58,69,6e,20,63,68,61,6f,21,00}

```

- Toán tử dán mã thông báo(##)(Token-pasting operator): sử dụng để tạo một mã thông báo mới từ 2 mã thông báo riêng biệt.Khi biên dịch 2 mã thông báo cần dán lại nằm ở 2 bên dấu (##)sẽ được ghép lại thành một mã thông báo mới liền nhau từ trái sang phải.

Ví dụ:

```

#define concat(a,b) (a##b)
...
char x,y,xy,z;
x=10;
y=20;
xy=30;
z=concat(x,y) //z=xy=30

```

❖ Câu hỏi ôn tập

1. Viết một CPP nếu biến Count chưa định nghĩa,đặt Count=20
2. Viết một CPP nếu biến IO_dr đã định nghĩa là input PortB,đặt IO_Port=PINB
3. Viết một macro ghép 2 biến độ dài byte thành 1 biến độ dài word.
4. Viết một macro cộng 2 số BCD kết quả trả về là số BCD
5. Viết một macro trả về thương số và dư số phép chia số nhị phân 16 bit cho 8 bit

5.13 Soạn thảo hợp ngữ trong C

Trong thực tế có những chương trình bắt buộc ta phải viết hợp ngữ vì các lý do sau:

- Tiết kiệm dung lượng bộ nhớ
- Cần biết và điều khiển chính xác thời gian thực
- Điều khiển xử lý bit
- Tổ chức quản lý bộ nhớ theo vùng địa chỉ cụ thể
- Can thiệp trực tiếp đến phần cứng

C-AVR cho phép ta soạn thảo hợp ngữ và C theo các cách như gọi hàm C trong chương trình hợp ngữ,goi chương trình con hợp ngữ trong chương trình C,soạn thảo hợp ngữ trong chương trình C như một dòng lệnh.Hai cách đầu đòi hỏi ta phải nắm thật rõ cách trình biên dịch C sử dụng các GPRs, như trình bày trong mục 5.2.6,chứa dữ liệu vào,dữ liệu trả về,dữ liệu trung gian...,để ghi/đọc và lưu trữ khi sử dụng,tránh bị mất dữ liệu!Do đó sẽ khó quản lý kiểm tra đối với chương trình dài,sử dụng nhiều dữ liệu xuất nhập.Cách thứ ba tỏ ra thích hợp hơn nhờ các ưu điểm sau:

- Không yêu cầu ta phải nhớ rõ cách trình biên dịch sử dụng thanh ghi
- Có thể chỉ định thanh ghi cho trình biên dịch
- Soạn thảo hợp ngữ như là một dòng phát biểu trong chương trình C
- Sử dụng chung các biến khai báo trong C

5.13.1 Định dạng một dòng hợp ngữ trong C

❖ Cú pháp:

```
asm ("code " : ouput op. list : input op. list [: clobber reg. list]);
```

asm là từ khóa báo dòng phát biểu hợp ngữ.

Dòng phát biểu có 4 trường,mỗi trường phân biệt với nhau bằng dấu (:) :

- code : gồm 1 hoặc nhiều lệnh hợp ngữ viết thành 1 dòng đặt toàn bộ trong dấu (“ ”)
- output op. list : danh sách các toán hạng ngõ ra,mỗi toán hạng cách nhau bởi dấu (,)
- input op. list : danh sách các toán hạng ngõ vào,mỗi toán hạng cách nhau bởi dấu (,)
- clobber reg. list : danh sách thanh ghi chỉ định cho trình biên dịch sử dụng,có thể bỏ qua

➤ *Ở đây ta hiểu toán hạng ngõ vào và toán hạng ngõ ra lần lượt chỉ các biến chứa dữ liệu nhập ,gọi tắt là các biến ngõ vào(input),và các biến chứa dữ liệu xuất,gọi tắt là các biến ngõ ra(output).Các biến*

này đã khai báo trong C. Trình biên dịch sẽ sử dụng các GPRs như mô tả trong mục 5.2.6 làm trung gian thực hiện các phát biểu khi biên dịch.

Ta lấy ví dụ đọc dữ liệu từ PortA cát vào biến val viết bằng dòng hợp ngữ như sau:

```
asm ("in %0,%1" : "=r" (val) : "I" (_SFR_IO_ADDR(PINA)));
```

- “in %0,%1” là dòng lệnh hợp ngữ đọc dữ liệu từ toán hạng %1 vào toán hạng %0.

Các ký hiệu %0,%1 biểu diễn các toán hạng theo số thứ tự, đầu tiên đánh số 0,kế tiếp là số 1 và tăng dần.

- “=r” (val) biểu diễn toán hạng ngõ ra là biến val(khai báo trong C)được cát trong GPR

- “I”(_SFR_IO_ADDR(PINA)) biểu diễn toán hạng ngõ vào là IOR.

(_SFR_IO_ADDR(PINA)) là macro của C trả địa chỉ PINA

- Dòng phát biểu trên không chỉ định sử dụng GPR.

- Do trình biên dịch C có công cụ tối ưu hóa biên dịch,có thể dẫn đến bỏ qua một số biến trung gian,giá trị trả về,nên đôi khi kết quả biên dịch có thể thực thi sai hoặc thậm chí bỏ qua dòng phát biểu hợp ngữ!Để đảm bảo biên dịch chính xác,bảo toàn các biến sử dụng,giá trị trả về,ta thêm từ khóa thuộc bộ định tính(qualifiers) **volatile** sau từ **khóa asm** của dòng phát biểu.

```
asm volatile("in %0,%1" : "=r" (val) : "I" (_SFR_IO_ADDR(PINA)));
```

Ví dụ 5.46: Xem đoạn chương trình sau và file.lss sau khi biên dịch:

```
#include <avr/io.h>
char val,dat_in,dat_out;
int main(void)
{
    DDRA=0x00;
    PORTA=0xff;
    while (1)
    {
        asm volatile("in %0,%1" : "=r" (val) : "I" (_SFR_IO_ADDR(PINA)));
    }
}
```

Sau khi biên dịch ra file.lss:

```
aa: 80 b1      in r24, 0x00 ; 0          //đọc data từ PINA
ac: 80 93 02 01 sts 0x0102, r24 ; 0x800102<val> //cất vào biến val
```

Trình biên dịch sử dụng r24 và gán vào biến val là ô nhớ SRAM địa chỉ 0x0102

- Trường hợp dòng phát biểu không có toán hạng nguồn,đích ta vẫn phải ghi dấu phân biệt trường (:)
- Ví dụ asm volatile (“nop” ::);

5.13.2 Trường lệnh “code”

- Là các mã lệnh như trong hợp ngữ AVR

- Các toán hạng được ký hiệu dấu % theo sau là số nguyên bắt đầu từ 0,tăng dần theo số toán hạng

- Quy định trường lệnh viết 1 dòng.Truường hợp trường lệnh gồm nhiều dòng lệnh,ta có thể dùng kết hợp ký hiệu LF \n và TAB \t vừa báo ngắt dòng và nối dòng để phân biệt giữa 2 lệnh.Ví dụ:

Trình hợp ngữ

nop

nop

nop

nop

Dòng hợp ngữ trong C

```
asm volatile ("nop\n\t"
            "nop\n\t"
            "nop\n\t"
            "nop\n\t"
            :: );
```

- Nhờ dấu \n\t ta có thể trình bày trường lệnh thành đoạn chương trình như trong lập trình hợp ngữ để dễ tham khảo,trình biên dịch C vẫn hiểu là 1 dòng phát biểu!

❖ Các thanh ghi đặc biệt

Ta có thể sử dụng các thanh ghi đặc biệt trực tiếp bằng các ký hiệu như sau:

Ký hiệu	Thanh ghi
SREG	IOR SREG cờ trạng thái
SPH	IOR SPH stack byte cao
SPL	IOR SPL stack byte thấp

temp reg	GPR r0 chứa dữ liệu tạm thời
zero reg	GPR r1 luôn bằng 0x00

5.13.3 Trường toán hạng ngõ vào và ngõ ra(input op. list,output op. list)

Mỗi toán hạng ngõ vào,ngõ ra được mô tả bằng một ký tự hạn định(constraints) trong dấu (" ") theo sau là biểu thức C trong dấu (.).Bảng sau mô tả các ký tự hạn định:

Ký tự hạn định	Sử dụng	Tầm
a	Nhóm GPR cao đơn giản	r16 - r23
b	Cặp thanh ghi con trỏ cơ bản	y,z
d	Nhóm GPR cao	r16 - r31
e	Cặp thanh ghi con trỏ	x,y,z
G	Hằng số dấu chấm động	0.0
I	Hằng số nguyên dương 6 bit	0 - 63
J	Hằng số nguyên âm 6 bit	-63 - 0
K	Hằng số nguyên	2
L	Hằng số nguyên	0
l	Nhóm GPR thấp	r0 - r15
M	Hằng số nguyên 8 bit	0 - 255
N	Hằng số nguyên	-1
O	Hằng số nguyên	8,16,24
P	Hằng số nguyên	1
p	Thanh ghi stack	SPH:SPL
r	GPR bất kỳ	r0 - r31
t	thanh ghi tạm	r0
w	Cặp thanh ghi cao đặc biệt	r24,r26,r28,r30
x	Cặp thanh ghi con trỏ	x(r27:r26)
y	Cặp thanh ghi con trỏ	y(r29:r28)
z	Cặp thanh ghi con trỏ	z(r31:r30)

➤ Lưu ý: Ta phải chọn chính xác ký tự hạn định cho toán hạng để khi biên dịch đúng cú pháp tập lệnh, bởi vì trình biên dịch C không hiểu các lệnh hợp ngữ nên vẫn biên dịch và có thể cho kết quả sai!

Ví dụ để viết lệnh OR biến val với 0xf0:

asm volatile("ori %0,%1" : "=d"(val) : "M"(0xf0));

Hoặc lệnh cộng biến val16 (16 bit) với 0x3a:

asm volatile("addiw %0,%1" :"=w"(val16) : "I"(0x3a);

Các ký tự hạn định có thể có các ký tự bổ nghĩa (modifiers) đứng trước.Bảng sau mô tả các ký tự bổ nghĩa:

Ký tự bổ nghĩa	Chỉ định
=	Toán hạng chỉ ghi,thường sử dụng cho tất cả toàn hạng đích(outputs)
+	Toán hạng đọc/ghi(không hỗ trợ cho dòng hợp ngữ)
&	Thanh ghi chọn sử dụng làm đích(outputs)

- Các ký tự hạn định không có ký tự bổ nghĩa đứng trước chỉ định toán hạng chỉ đọc
- Toán hạng ngõ ra là toán hạng chỉ ghi
- Toán hạng ngõ vào là toán hạng chỉ đọc
- Biểu thức C phải hợp lệ giữa toán hạng với giá trị gán bên trái(lvalue)
- Danh sách các toán hạng ngõ ra và ngõ vào sắp xếp theo thứ tự truy xuất và ghi theo ký hiệu đã gán trong trường lệnh “code”.

1. Toán hạng vừa là ngõ vào,vừa là ngõ ra

Toán hạng có thể vừa là ngõ vào,vừa là ngõ ra,như lệnh SWAP Rd.Trong trường hợp này ,ta thay ký tự hạn định ở trường toán hạng ngõ vào bằng số thứ tự toán hạng giống như số thứ tự đặt cho toán hạng,để báo trình biên dịch sử dụng cùng một thanh gh

Ví dụ lệnh SWAP val:

asm volatile("%0" : "=r" (val): "0"(val));

Một trường hợp sử dụng toán hạng vừa là ngõ vào ở một lệnh,vừa là ngõ ra ở lệnh sau đó trong hợp

ngữ. Ví dụ đọc PORTB cát vào biến dat_in,sau đó ghi biến dat_out ra PORTB,nếu viết dòng hợp ngữ như sau:

```
asm volatile(
    "in %0,%1 \n\t"
    "out %1,%2 \n\t"
    :"=r"(dat_in)
    :"I"(_SFR_IO_ADDR(PORTB)),"r"(dat_out)
);
```

Sau khi biên dịch cho kết quả:

```
asm volatile(
ae: 80 91 01 01 lds r24, 0x0101 ; 0x800101 <dat_out>
b2: 85 b1      in  r24, 0x05   ; 5
b4: 85 b9      out 0x05, r24  ; 5
b6: 80 93 00 01 sts 0x0100, r24 ; 0x800100 <__DATA_REGION_ORIGIN__>
```

Nội dung dat_out cát trong r24 bị mất do lệnh in đầu tiên,gia trị ghi ra PortB sau đó chính là nội dung đọc từ Port vào(dat_in).Điều này xảy ra do trình biên dịch sử dụng cùng một thanh ghi cho toán hạng ngõ vào và ngõ ra.Để tránh tình trạng trên,ta chỉ cần sử dụng ký tự bô nghĩa & chỉ định thanh ghi chứa biến dat_in là ngõ ra chỉ ghi:

```
asm volatile(
    "in %0,%1 \n\t"
    "out %1,%2 \n\t"
    :"=&r"(dat_in)
    :"I"(_SFR_IO_ADDR(PORTB)),"r"(dat_out)
);
```

Sau khi biên dịch cho kết quả:

```
asm volatile(
ae: 90 91 01 01 lds r25, 0x0101 ; 0x800101 <dat_out>
b2: 85 b1      in  r24, 0x05   ; 5
b4: 95 b9      out 0x05, r25  ; 5
b6: 80 93 00 01 sts 0x0100, r24 ; 0x800100 <__DATA_REGION_ORIGIN__>
```

Ví dụ 5.47: Ví dụ sau đây sử dụng 2 biến dat_out và val để đảo bit PB0 .Trước tiên nạp dat_out=0x01,đảo bit PB0 và sau đó dịch trái dat_out,lưu giá trị xuất ra PortB vào biến val khi thực hiện xong

```
asm volatile(
    "ldi %0,0x01 \n\t"
    "in %1,%2 \n\t"
    "eor %1,%0 \n\t"
    "out %2,%1 \n\t"
    "lsl %0"
    :"=&d"(dat_out),"=&r"(val)
    :"I"(_SFR_IO_ADDR(PORTB))
);
```

- Ở đây ta có hai biến ngõ ra phân biệt dat_out chứa giá trị 0x01 dịch trái 1 lần,và val chứa giá trị xuất ra PortB,nên khai báo các biến ngõ ra "=&d"(dat_out),và "=&r"(val)

- Toán hạng PORTB (gọi macro) chỉ định bằng từ hạn định “I”(IOR địa chỉ từ 0-63)là biến ngõ vào.
- Danh sách các ngõ ra truy xuất theo thứ tự dat_out,val
- Danh sách các ngõ vào chỉ có PORTB

2. Biểu diễn byte cao,byte thấp của một biến

Để biểu diễn các byte của một biến ví dụ 4 byte,ta sử dụng các ký hiệu %A0 cho byte thấp,%B0 cho byte thứ hai,%C0 cho byte thứ ba và %D0 cho byte thứ tư,đối với biến đầu tiên số thứ tự 0.Tương tự biến thứ hai là %A1,%B1,%C1,%D1 theo thứ tự từ byte thấp đến cao...

Ví dụ 5.48: Dòng hợp ngữ sau thực hiện việc swap một biến 16 bit val16:

```
asm volatile(
    "mov __tmp_reg__,%A0 \n\t" //r0=val16L
    "mov %A0,%B0          \n\t" //val16L=val16H
    "mov %B0,__tmp_reg__ \n\t" // val16H=r0
```

```

        :"=r"(val16)
        :"0"(val16)
    );

```

Biến val16 vừa là ngõ vào,vừa là ngõ ra nên ta ký hiệu số thứ tự toán hạng là 0 thay cho từ hạn định khi khai báo toán hạng ngõ vào. Biến val16 gồm 2 byte, byte thấp ký hiệu %A0,byte cao %B0.Ta sử dụng thanh ghi tạm r0 _temp_reg_. Trình biên dịch sẽ gán %A0=r24,%B0=r25.

3. Sử dụng thanh ghi con trỏ x,y,z

Ví dụ 5.49: Ta xem ví dụ sau khởi động xóa vùng nhớ SRAM bắt đầu từ địa chỉ cát trong biến temp16(16 bit),chiều dài cát trong biến count(8 bit),biến val8=0x00.

```

asm volatile(
    "ldi %0,10          \n\t"
    "ldi %1,0x00         \n\t"
    "loop: st x+,%1      \n\t"
    "dec %0              \n\t"
    "brne   loop         \n\t"
    :
    :"d"(count),"d"(val8),"x"(temp16)
);

```

Sau khi biên dịch cho kết quả:

```

asm volatile(
a4:  80 91 03 01 lds r24, 0x0103 ; 0x800103 <count>
a8:  90 91 00 01 lds r25, 0x0100 ; 0x800100 <__DATA_REGION_ORIGIN__>
ac:  a0 91 01 01 lds r26, 0x0101 ; 0x800101 <temp16>
b0:  b0 91 02 01 lds r27, 0x0102 ; 0x800102 <temp16+0x1>
b4:  8a e0          ldi r24, 0x0A ; 10
b6:  90 e0          ldi r25, 0x00 ; 0

000000b8 <loop>:
b8:  9d 93          st X+, r25
ba:  8a 95          dec r24
bc:  e9 f7          brne.-6 ; 0xb8 <loop>

```

Ta có thể viết lại dòng hợp ngữ trên như sau:

```

asm volatile(
    "ldi r17,10          \n\t"
    "ldi r16,0x00         \n\t"
    "loop: st %a0+,r16    \n\t"
    "dec r17              \n\t"
    "brne   loop         \n\t"
    :
    :"e"(temp16)
);

```

Sau khi biên dịch cho kết quả:

```

asm volatile(
a4:  e0 91 01 01 lds r30, 0x0101 ; 0x800101 <temp16>
a8:  f0 91 02 01 lds r31, 0x0102 ; 0x800102 <temp16+0x1>
ac:  1a e0          ldi r17, 0x0A ; 10
ae:  00 e0          ldi r16, 0x00 ; 0

000000b0 <loop>:
b0:  01 93          st Z+, r16
b2:  1a 95          dec r17
b4:  e9 f7          brne.-6 ; 0xb0 <loop>

```

Ta sử dụng r17 thay cho biến count,r16 thay cho biến val8.Nếu biến vào temp16(16 bit) khai báo ký tự hạn định “e” chỉ con trỏ địa chỉ chung(không phải khai báo cụ thể x,y,z),trình biên dịch sẽ tự động gán con trỏ địa chỉ Z.Trong phần toán hạng chỉ con trỏ trong trường lệnh,tu có thể ký hiệu z hay %a0(ký hiệu %A0 sẽ bị báo lỗi!)

5.13.4 Trường thanh ghi chỉ định(clobber reg. list)

Trường cuối cùng trong dòng lệnh hợp ngữ là trường chỉ định thanh ghi cho trình biên dịch.Thông thường ta có thể bỏ qua trường này.Tuy nhiên trong một số trường hợp,để thuận lợi trong việc soạn thảo dòng hợp ngữ,ta sử dụng các thanh ghi cụ thể và để tránh cho trình biên dịch sử dụng các thanh ghi này lưu và chuyển các thông số hình thức trong lúc biên dịch,ta phải liệt kê danh sách các thanh ghi chỉ định sắp xếp theo thứ tự sử dụng ở trường này,mỗi ký hiệu thanh ghi đặt trong dấu (" "),cách nhau dấu (,).

Ví dụ 5.50: Trở lại ví dụ 5.49,ta thay các thanh ghi r17,r16 bằng r24,r25,gán biến count chứa chiều dài vùng nhớ vào r24.Ta khai báo r24,r25 ở trường các thanh ghi chỉ định.

asm volatile(

```
"mov r24,%0          \n\t"
"ldi r25,0x00        \n\t"
"loop: st %a1+,r25  \n\t"
"dec r24             \n\t"
"brne   loop         \n\t"
:
:"d"(count),"e"(temp16)
:"r24","r25"
);
```

Kết quả sau khi biên dịch:

```
asm volatile(
a4: 20 91 03 01 lds r18, 0x0103 ; 0x800103 <count>
a8: e0 91 01 01 lds r30, 0x0101 ; 0x800101 <temp16>
ac: f0 91 02 01 lds r31, 0x0102 ; 0x800102 <temp16+0x1>
b0: 82 2f          mov r24, r18
b2: 90 e0          ldi r25, 0x00 ; 0
000000b4 <loop>:
b4: 91 93          st Z+, r25
b6: 8a 95          dec r24
b8: e9 f7          brne.-6 ; 0xb4 <loop>
```

Trình biên dịch sẽ thay toán hạng %0 bằng r18 và chuyển biến count vào r18.Nếu không khai báo các thanh ghi chỉ định,trình biên dịch sẽ thay toán hạng %0 bằng r24,bên dịch lệnh mov r24,%0 thành mov r24,r24,trong một số trường hợp chương trình sẽ bị sai!!!

➤ Giá trị biến temp16 tự động cập nhật theo giá trị con trỏ địa chỉ khi thoát khỏi dòng hợp ngữ.Trong trường hợp ta muốn giữ nguyên giá trị biến temp16 ngay trước khi bắt đầu dòng hợp ngữ,ta thêm chỉ định thanh ghi "memory" báo trình biên dịch không thay đổi giá trị biến temp16 khi tiếp tục sử dụng nó tiếp theo dòng hợp ngữ.Ta xem ví dụ minh họa sau đây:

Ví dụ 5.51: Đoạn chương trình sau không khai báo thanh ghi chỉ định "memory":

```
count=10;
temp16=0x110;
while (1)
{
    asm volatile(
        "mov r24,%0          \n\t"
        "ldi r25,0x00        \n\t"
        "loop: st %a1+,r25  \n\t"
        "dec r24             \n\t"
        "brne   loop         \n\t"
        :
        :"d"(count),"e"(temp16)
        :"r24","r25"
    );
    temp16+=0x20;
}
```

Kết quả sau khi biên dịch:

```
count=10;
a4: 8a e0          ldi r24, 0x0A ; 10
a6: 80 93 03 01 sts 0x0103, r24 ; 0x800103 <count>
```

```

temp16=0x110;
aa: 80 e1      ldi r24, 0x10 ; 16
ac: 91 e0      ldi r25, 0x01 ; 1
ae: 90 93 02 01 sts 0x0102, r25 ; 0x800102 <temp16+0x1>
b2: 80 93 01 01 sts 0x0101, r24 ; 0x800101 <temp16>
while (1)
{
    asm volatile(
b6: e0 91 01 01 lds r30, 0x0101 ; 0x800101 <temp16>
ba: f0 91 02 01 lds r31, 0x0102 ; 0x800102 <temp16+0x1>
be: 2a e0      ldi r18, 0x0A ; 10
c0: 82 2f      mov r24, r18
c2: 90 e0      ldi r25, 0x00 ; 0

000000c4 <loop>:
c4: 91 93      st Z+, r25
c6: 8a 95      dec r24
c8: e9 f7      brne.-6 ; 0xc4 <loop>
    "brne loop \n\t"
    :
    :"d"(count), "e"(temp16)
    :"r24", "r25"
);
temp16+=0x20;
ca: b0 96      adiwr30, 0x20 ; 32
cc: f0 93 02 01 sts 0x0102, r31 ; 0x800102 <temp16+0x1>
d0: e0 93 01 01 sts 0x0101, r30 ; 0x800101 <temp16>
d4: f0 cf      rjmp.-32 ; 0xb6 <main+0x12>

```

Sau khi thoát khỏi dòng hợp ngữ Z=0x11B, phát biểu temp+=0x20 được biên dịch lấy Z cộng thêm 0x20, như vậy temp16=0x13B.

Bây giờ ta thêm “memory” vào trường thanh ghi chỉ định:

```

count=10;
temp16=0x110;
while (1)
{
    asm volatile(
        "mov r24,%00      \n\t"
        "ldi r25,0x00      \n\t"
        "loop; st %al+,r25 \n\t"
        "dec r24          \n\t"
        "brne loop         \n\t"
        :
        :"d"(count), "e"(temp16)
        :"r24", "r25", "memory"
    );
    temp16+=0x20;
}

```

Kết quả sau khi biên dịch:

```

count=10;
a4: 8a e0      ldi r24, 0x0A ; 10
a6: 80 93 03 01 sts 0x0103, r24 ; 0x800103 <count>
temp16=0x110;
aa: 80 e1      ldi r24, 0x10 ; 16
ac: 91 e0      ldi r25, 0x01 ; 1
ae: 90 93 02 01 sts 0x0102, r25 ; 0x800102 <temp16+0x1>
b2: 80 93 01 01 sts 0x0101, r24 ; 0x800101 <temp16>
while (1)
{
    asm volatile(
b6: 20 91 03 01 lds r18, 0x0103 ; 0x800103 <count>
ba: e0 91 01 01 lds r30, 0x0101 ; 0x800101 <temp16>
be: f0 91 02 01 lds r31, 0x0102 ; 0x800102 <temp16+0x1>
c2: 82 2f      mov r24, r18

```

```

c4: 90 e0      ldi r25, 0x00 ; 0
000000c6 <loop>:
c6: 91 93      st Z+, r25
c8: 8a 95      dec r24
ca: e9 f7      brne.-6 ; 0xc6 <loop>
                "brne loop \n\t"
                :
                :"d"(count), "e"(temp16)
                :"r24", "r25", "memory"
                );
temp16+=0x20;
cc: 80 91 01 01 lds r24, 0x0101 ; 0x800101 <temp16>
d0: 90 91 02 01 lds r25, 0x0102 ; 0x800102 <temp16+0x1>
d4: 80 96      adiwr24, 0x20 ; 32
d6: 90 93 02 01 sts 0x0102, r25 ; 0x800102 <temp16+0x1>
da: 80 93 01 01 sts 0x0101, r24 ; 0x800101 <temp16>
de: eb cf      rjmp.-42 ; 0xb6 <main+0x12>

```

Ở phát biểu temp16+=0x20, trình biên dịch lấy giá trị biến ban đầu temp16=0x110 cộng 0x20, kết quả temp16=0x130.

Một phương pháp khác có thể giữ được giá trị biến temp16 không bị cập nhật theo dòng lệnh hợp ngữ mà không cần khai báo “memory”, là ta sẽ khai báo thêm biến con trả để trả vào biến temp16, ví dụ như:

```

int *ptr16;
...
count=10;
temp16=0x110;
*ptr16=temp16;
while (1)
{
    asm volatile(
        "mov r24,%0\n\t"
        "ldi r25,0x00\n\t"
        "loop: st %a1+,r25\n\t"
        "dec r24\n\t"
        "brne loop\n\t"
        :
        :"d"(count), "e"(ptr16)
        :"r24", "r25"
    );
    temp16+=0x20;
}

```

Kết quả thực hiện đoạn chương trình trên temp16=0x130, ptr=0x13B.

5.13.5: Macro hợp ngữ

Ta có thể soạn thảo dòng hợp ngữ dưới dạng macro hợp ngữ và lưu trong thư viện để tiện sử dụng. Khi gọi các macro sử dụng CPP #include. Trong thư viện có lưu nhiều macro hợp ngữ đã soạn thảo sẵn. Để tránh tình trạng trùng hợp và thuận tiện khi biên dịch, ta nên thay từ khóa asm bằng __asm__ và volatile bằng volatile .

Định dạng khai báo macro hợp ngữ như sau:

#define Tên_macro (danh sách thông số hình thức) (thân macro)

- Tên_macro tương tự như danh hiệu
- Danh sách thông số hình thức tương tự như macro trong C, không cần khai báo kiểu dữ liệu
- Thân macro tương đương như dòng hợp ngữ bắt đầu bằng __asm__ __volatile__ (...)
- Kết thúc macro không có dấu ;)
- Thân macro phải cùng dòng với Tên macro và (danh sách thông số hình thức)
- Ta sử dụng toán tử nối dòng (\) trong trường hợp muốn xuống dòng

Một vấn đề nữa trong soạn thảo thân macro khi có sử dụng nhãn. Cách sử dụng nhãn trong dòng hợp ngữ hoàn toàn tương tự như hợp ngữ, chỉ thêm ký tự %= sau tên nhãn theo định dạng:

tên_nhãn%=[tên nhãn mở rộng]:

Ta xem ví dụ sau đây, viết một macro thực hiện chờ đến khi bit port bị xóa sẽ thoát.

Ví dụ 5.52:

```
#include <avr/io.h>
#define loop_sbic(port,bit) \
    __asm__ __volatile__ ( \
        "L_%=: sbic %0,%1          \n\t" \
        "rjmp  L_%=                \n\t" \
        : \
        :"I"(_SFR_IO_ADDR(port)), "I"(bit) \
    )

int main(void)
{
    DDRB=0xf0;//PB0..PB3 input,PB4..PB7 output
    PORTB=0x0f;//R kéo lên PB0..PB3,PB4:PB7=0
    while (1)
    {
        loop_sbic(PINB,0); //chờ PB0=0
        PORTB|=(1<<PB7); //đặt PB7=1
        loop_sbic(PINB,1); //chờ PB1=0
        PORTB&=~(1<<PB7); //xóa PB7=0
    }
}
```

Kết quả sau khi biên dịch:

```
int main(void)
{
    DDRB=0xf0;
94:  80 ef      ldi r24, 0xF0 ; 240
96:  84 b9      out 0x04, r24 ; 4
    PORTB=0x0f;
98:  8f e0      ldi r24, 0x0F ; 15
9a:  85 b9      out 0x05, r24 ; 5

0000009c <L_11>:
    while (1)
    {
        loop_sbic(PINB,0);
9c:  18 99      sbic0x03, 0 ; 3
9e:  fe cf      rjmp .-4      ; 0x9c <L_11>
    PORTB|=(1<<PB7);
a0:  85 b1      in  r24, 0x05 ; 5
a2:  80 68      ori r24, 0x80 ; 128
a4:  85 b9      out 0x05, r24 ; 5

000000a6 <L_15>:
        loop_sbic(PINB,1);
a6:  19 99      sbic0x03, 1 ; 3
a8:  fe cf      rjmp .-4      ; 0xa6 <L_15>
    PORTB&=~(1<<PB7);
aa:  85 b1      in  r24, 0x05 ; 5
ac:  8f 77      andi r24, 0x7F ; 127
ae:  85 b9      out 0x05, r24 ; 5
b0:  f5 cf      rjmp .-22     ; 0x9c <L_11>
```

Ở lần gọi macro thứ nhất nhãn được gán L_11, ở lần gọi macro thứ hai nhãn được gán L_15 (ký hiệu thêm vào phân biệt tên nhãn khi gọi macro nhiều lần do trình biên dịch tự đặt).

5.13.6 Hàm trong C gắn dòng hợp ngữ

Sử dụng macro hợp ngữ trong C có điểm thuận lợi là không cần định nghĩa trước thông số hình thức và khi biên dịch cho kết quả hoàn toàn tương tự như các lệnh hợp ngữ mô tả trong macro. Tuy nhiên khi sử dụng macro nhiều lần sẽ có điểm bất lợi là dung lượng chương trình tăng, nhất là khó kiểm soát các

giá trị biến cần lưu sau khi thoát khỏi macro. Để tránh những bất lợi trên, ta sử dụng hàm khai báo trong C nhưng định nghĩa hàm bằng dòng hợp ngữ, hoàn toàn có thể kiểm soát được các biến trong C. Trình biên dịch thực hiện hoàn toàn như hàm bình thường viết bằng ngôn ngữ C.

Ví dụ 5.53: Ta tạo hàm tạo trễ đơn vị ms có thể thay đổi từ 1 - 255ms, không phụ thuộc vào tần số Fck. Áp dụng hàm tạo trễ trên viết chương trình C tạo chuỗi xung vuông đối xứng chu kỳ 10ms trên PB0.

Giải:

```
#include <avr/io.h>
#define clock 8000          //Fck tính bằng KHz
unsigned int ct_delay;
void delay_ms(unsigned char n); //khai báo hàm delay_ms

int main(void)
{
    DDRB|=(1<<PB0);      //PB0 output
    PORTB&=~(1<<PB0);    //xóa PB0=0
    ct_delay=clock/4;       //giá trị ct_delay tương đương 1ms
    while (1)
    {
        PORTB^=(1<<PB0); //đảo bit PB0
        delay_ms(5);        //delay 5ms
    }
}
void delay_ms(unsigned char n)
{
    unsigned int cnt;
    asm volatile (
        "L%0=1: mov %A0,%A1 \n\t" //nạp ct_delay
        "mov %B0,%B1 \n\t"
        "L%0=2: sbiw %A0,1 \n\t" //trù 1.2MC
        "brne L%0=2 \n\t" //lặp vòng nếu khác 0 2/1MC
        "dec %2 \n\t" //đếm số vòng delay ms
        "brne L%0=1 \n\t"
        :"=&w"(cnt)           //biến đếm
        :"r"(ct_delay),"r"(n)   //biến cần trả về
    );
}
```

Hàm delay_ms khai báo bằng C nhưng định nghĩa bằng dòng hợp ngữ. Để tạo một đơn vị thời gian trễ bằng 1ms, vòng lặp L%0=2 gồm 2 lệnh sbiw và brne mất 4MC sẽ lập vòng số lần bằng giá trị nạp vào biến ct_delay. Giá trị nạp vào biến ct_delay bằng tần số Fck chia 4, với Fck=8000Khz, ct_delay=2000.

Thông số hình thức n là số lần lặp vòng ms hay là số ms cần tạo trễ. Ở đây ta khai báo ct_delay là biến toàn cục, được nạp giá trị cố định đầu chương trình, biến đếm vòng lặp ms là biến cục bộ cnt, nên khi thoát khỏi dòng hợp ngữ giá trị biến ct_delay vẫn giữ nguyên.

❖ Kết quả trả về

Hàm trong C có gắn dòng hợp ngữ vẫn sử dụng kết quả trả về bằng phát biểu return như bình thường. Ta xem ví dụ minh họa sau đây.

Ví dụ 5.54: Hàm trong C sau đây ghép byte cao và byte thấp là thông số hình thức, trả về biến 2 byte.

```
uint16_t inwp(uint8_t byte_h,uint8_t byte_l)
{
    uint16_t portw;
    asm volatile(
        "mov %A0,%1 \n\t"
        "mov %B0,%2 \n\t"
        :"=&r"(portw)
        :"r"(byte_l),"r"(byte_h)
```

```
    );
    return portw;
}
```

Kiểu dữ liệu uint16_t,uint8_t xác định chính xác biến dài 16 bit và 8 bit. Phát biểu return portw báo trình biên dịch trả về giá trị biến cục bộ portw 16 bit khi thoát khỏi hàm inpw.

Ví dụ để ghép 2 byte ngõ vào PortD và PortC(giả sử đã khai báo hợp lệ các biến), ta gọi hàm inwp:
val16=inwp(PIND,PINC);

❖ **Câu hỏi ôn tập**

1. Viết một dòng hợp ngữ trong C, nếu PB0=0 xuất 0xaa ra PortC, nếu PB0=1 xuất 0x55 ra PortC.
2. Viết một dòng hợp ngữ trong C đặt chuỗi ký tự “Hello! “ tại địa chỉ đầu SRAM=0x200.
3. Viết một macro dịch một biến a 8 bit ra PC0, MSB xuất trước, sử dụng dòng hợp ngữ trong C.
4. Viết một hàm delay từ 10 -1000μs càng chính xác càng tốt, biến vào hàm bằng thời gian delay.
5. Viết một hàm tạo 1 xung tích cực mức thấp trên 1 chân Port, thời đoạn xung thay đổi từ 10-1000μs áp dụng hàm delay câu 4.

BÀI TẬP CHƯƠNG 5

❖ Trong các bài tập sau cho $F_{osc}=F_{ck}=8MHz$, trừ trường hợp có ghi chú cụ thể

1. Viết một đoạn lệnh thực hiện:
 - (a) Dịch trái 1 số nhị phân 16 bit R21:R20(R21 byte cao)
 - (b) Quay trái 1 số nhị phân 16 bit R21:R20
2. Viết một đoạn lệnh nhận dạng cạnh xuống xuất hiện trên PB1.
3. Viết một đoạn chương trình chuyển 1 chuỗi data có ký tự kết thúc chuỗi=\$00 cát trong bộ nhớ Flash địa chỉ đầu=TAB,sang vùng nhớ SRAM địa chỉ đầu=S_BUFS(Các ký hiệu đã định nghĩa trước)
4. Viết một chương trình đọc data từ PORTD,nếu data=\$20 - \$7F xuất ra PORTB,nếu data <\$20 hoặc >\$7F xuất ra PORTB=\$00.Thực hiện lập vòng liên tục.
5. Viết một chương trình đọc byte data từ PORTA,dò từ LSB nếu gặp bit0 đầu tiên xuất bit0 đúng vị trí trọng số ra PORTC,nếu không có bit0,xuất \$FF ra PORTC,lập vòng liên tục.
6. Viết một hàm tên COMP16 so sánh 2 số nhị phân 16 bit cát trong 2 biến a và b,kết quả trả về bit cờ com_flg= 1 nếu a < b,com_flg=0 nếu a ≥ b,bảo toàn nội dung các biến.
7. Viết một hàm tên DIV16_8 chia số nhị phân 16 bit cho 8 bit,trả về thương số và dư số phép chia.
8. Viết một chương trình con tên MUL16_8,nhân số nhị phân 16 bit cho 8 bit,trả về kết quả 24 bit đặt trong 2 biến 16 bit.
9. Viết một đoạn chương trình thực hiện phép tính: $y=x^2-x+1, x=8 \text{ bit}, y=16 \text{ bit}$
10. Viết một hàm/macro tên BCD_HEX8 chuyển số BCD thành số HEX:input (BCD)=00-99, output (HEX)=\$00-\$63.
11. Viết một hàm/macro tên ADD_BCD cộng 2 số BCD(00-99),trả về kết quả số BCD(00-99)và 1 bit cờ báo tràn
12. Viết một hàm/macro tên SUB_BCD trừ 2 số BCD(00-99),trả về kết quả số BCD(00-99)và bít cờ báo tràn (âm)
13. Viết một hàm tạo thời gian trễ từ 10 - 1000μs(càng chính xác càng tốt),có biến nhập thay đổi thời gian.
14. Viết một chương trình tạo chuỗi xung vuông đối xứng tần số 1Khz,xuất ra PB5.
15. Viết một chương trình tạo chuỗi xung vuông tần số 1Khz,CKNV=30%,xuất ra PD7.