致尊敬的顾客

关于产品目录等资料中的旧公司名称

NEC电子公司与株式会社瑞萨科技于2010年4月1日进行业务整合（合并），整合后的新公司暨“瑞萨电子公司”继承两家公司的所有业务。因此，本资料中虽还保留有旧公司名称等标识，但是并不妨碍本资料的有效性，敬请谅解。

瑞萨电子公司网址：http://www.renesas.com

2010年4月1日
瑞萨电子公司

【发行】瑞萨电子公司（http://www.renesas.com）

【业务咨询】http://www.renesas.com/inquiry

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

740 族

编程指南<C 语言篇>

说明

本编程指南以应用于瑞萨8位单片机740族的C编译器M3T-ICC740为例，从C语言基础到ROM化进行了详细介绍。
可将本资料作为C语言入门以及编程时的参考资料使用。

另外,有关740族的各种硬件以及开发支持工具,可参考各用户使用说明及操作说明书。

本资料的使用方法

本资料是740族用C编译器M3T-ICC740的编程指南。
为更好地理解本资料内容,需事先熟悉740族产品结构以及汇编语言的相关知识。
本资料由两章构成。根据使用目不同请分别选择内容。
○C语言初学者→ 从第 1 章开始
○希望了解ICC740扩展功能→ 从第 2 章开始
本资料采用大内存模式(-ml操作)描述。
操作请参照ICC编译器编程指南(icc740_jp.pdf)。

本资料的最新版请在瑞萨科技网页上下载。

请确认最新版本后使用。

目录

第 1 章
C语言入门

本章向初次使用C语言的读者介绍了编写嵌入程序所必须的基础知识。

1.1 使用 C 语言编程

1.1.1 汇编语言和C语言

下面介绍C语言的主要特点以及如何使用C语言进行编程。

C 语言的特点

(1) 程序处理流程易懂.
基本结构中的顺序处理,分支处理及循环处理均可使用控制语句进行描述。因此,使用 C 语言比较容易读懂程序处理流程。

(2) 易于分成几个模块
使用 C 语言编写程序的基本单位被称为函数。因函数的参数独立性高,便于软件包重复利用。
另外,用汇编语言编写的模块可反复利用。

(3) 编写的程序程序易于维护
根据理由(1)和(2),容易进行运行后的程序维护。而且,因为 C 语言的标准规格(ANSI 规格(注))以被确定,源程序稍做修改便可插入其他的机型程序中使用。

(注) 为保持 C 语言的移植性,由 ANSI(American National Standards Institute)规定的 C 语言标准规格。

C 语言与汇编语言的比较

以下总结了源程序的描述方法以及与汇编语言的比较。

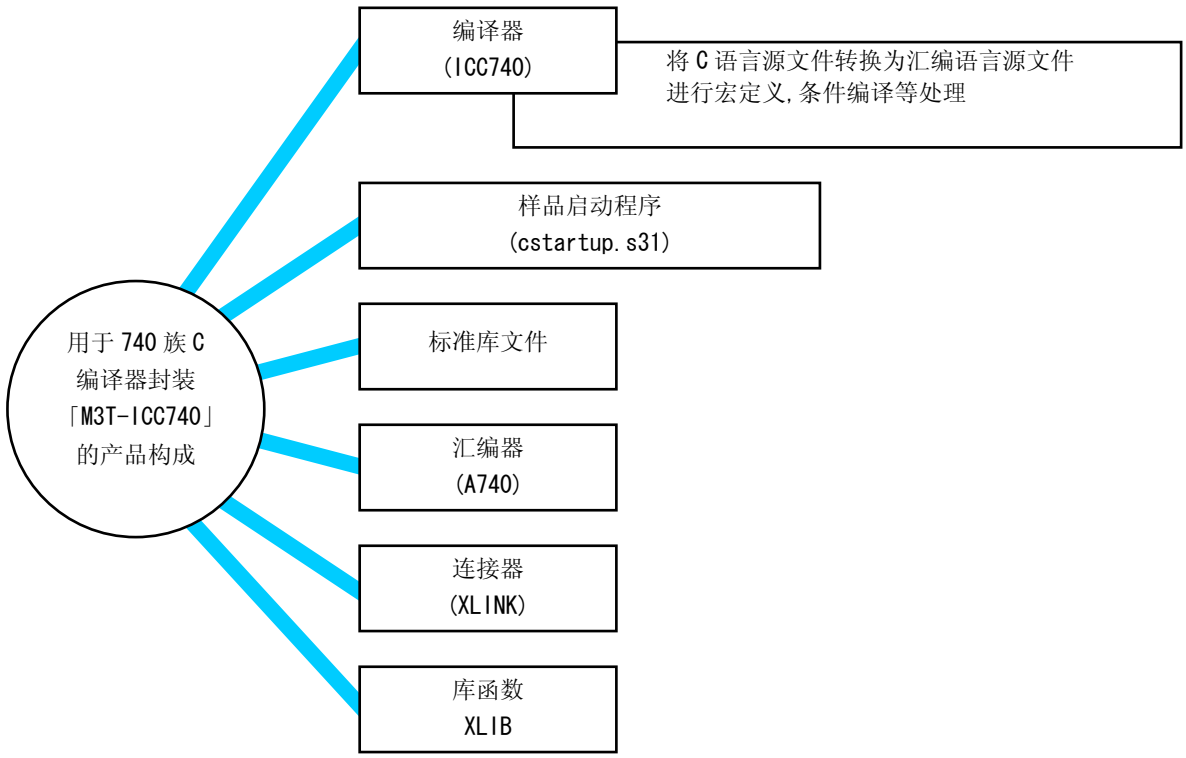
	C 言語	汇编语言
程序的基本单位 (描述方法)	函数 (函数名( ){ })	子程序 (子程序名:)
格式	自由格式,遵循 ANSI C 语言标准	一行一个命令
大写字母/小写字母的区别	区分大写字母与小写字母	不区分
分配数据空间	用类型指定	用尺寸(字节数)指定 (使用伪命令)

1.1.2 程序开发流程

将 C 语言描述的源程序转换成机器码的过程被称为「编译」。完成这个转换的软件被称为编译器。
在此介绍一下瑞萨 8 位单片机 740 族的 C 编译器封装「M3T-ICC740」的程序开发顺序。

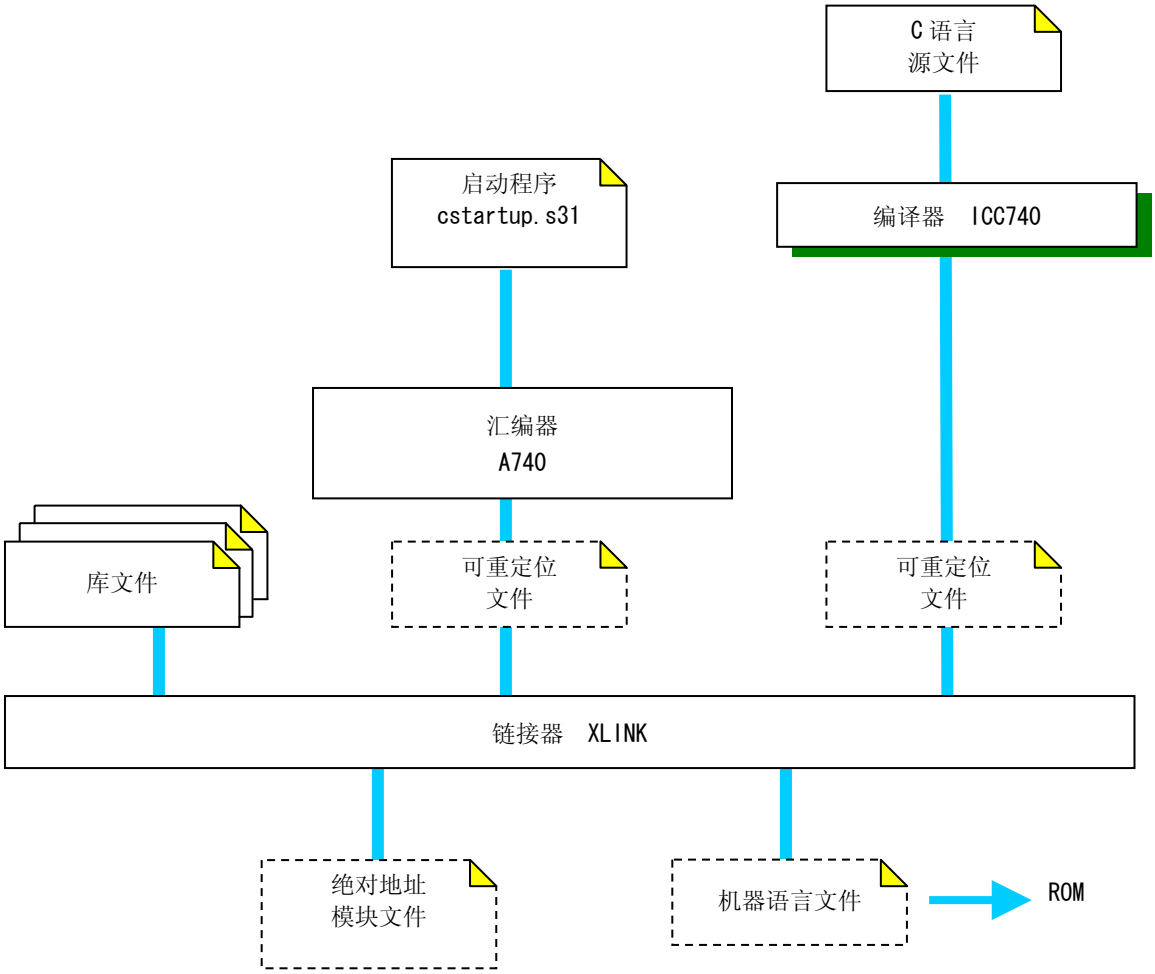
用于 740 族 C 汇编器封装「M3T-ICC740」的产品一览表

瑞萨 8 位单片机 740 族的 C 编译器 M3T-ICC740 中所包含的产品如下所示。



从源文件转换为机器码文件的过程

ICC740 中，要生成机器码文件，除了用 C 语言描述的源文件程序之外，还需要用汇编语言描述的启动程序。
形成机器码文件的工具包如下所示。



1.1.3 C程序规范

由于 C 语言程序格式自由,在遵循一定规则的基础上可以自由描述。编程要求简单明了且易于维护,不论是谁任何时候都能读懂。

下面介绍程序编写简明易懂的几个要点。

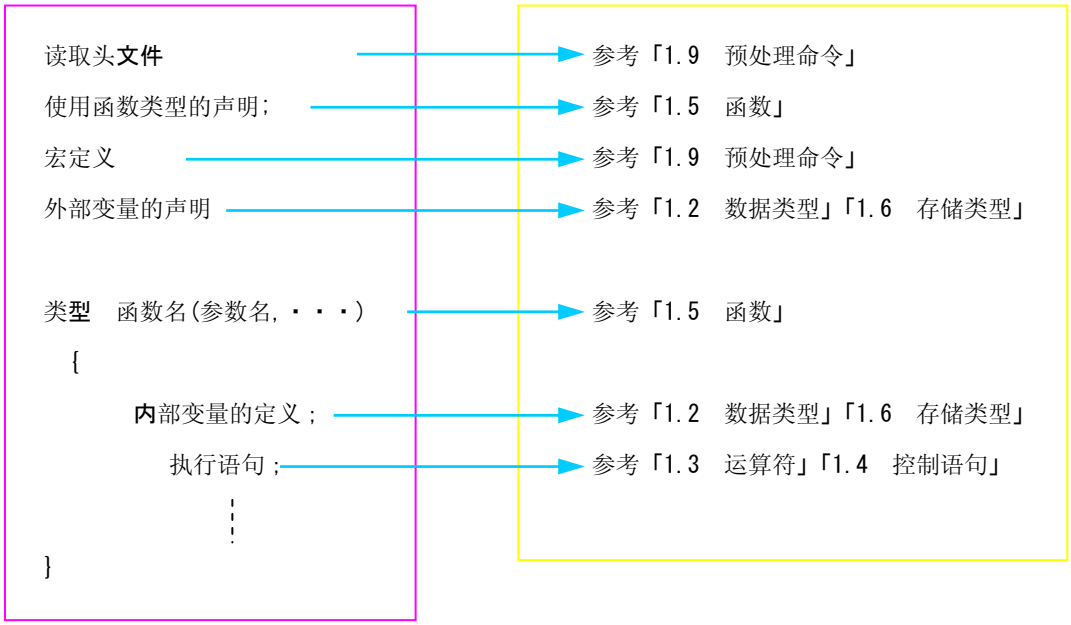
C 语言的规则

用 C 语言编写程序时,必须遵守下述 5 项规则。

(1) 各执行语句用分号「;」隔开。
(2) 函数执行语句和控制语句用大括号「{」「}」括起来。
(3) 函数和变量要有类型声明。
(4) 不使用保留字作变量名或函数名。
(5) 注释以「/* 注释 */」或、「// 注释」(C++形式)来记述。采用后者记述时,需要「-K」操作。

C 语言源文件的构成

以下是一般情况下 C 语言源文件的构成。对各项目请参考相应章节的内容。

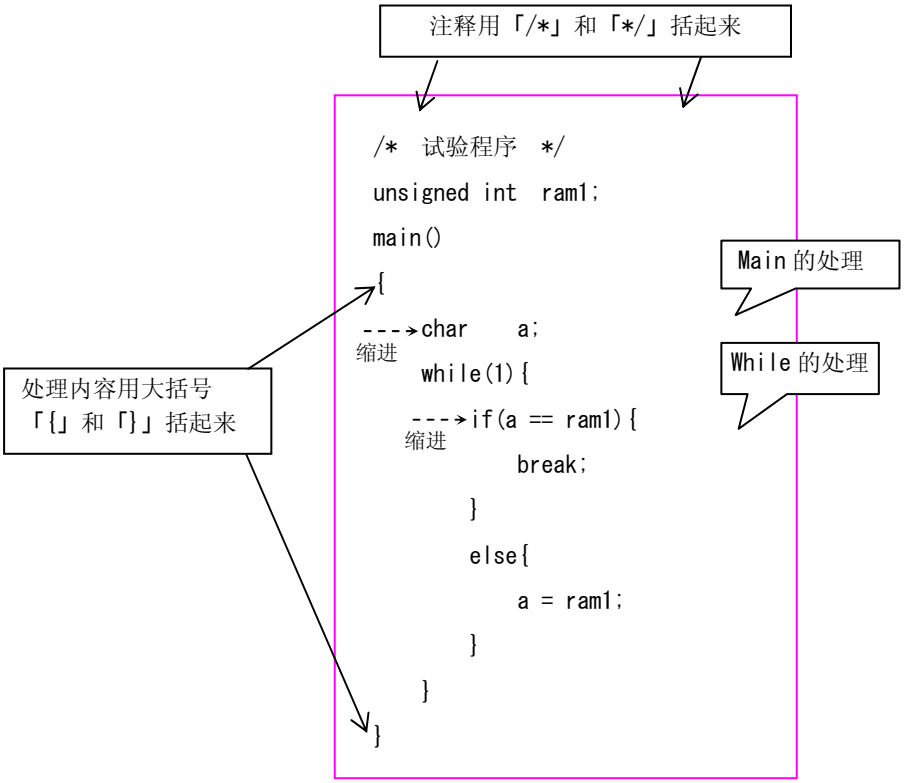


程序格式

为了使程序易读和维护，程序开发人员之间规定建立一个程序模板，作为共有程序格式，使每个人都易读懂且容易进行程序维护。

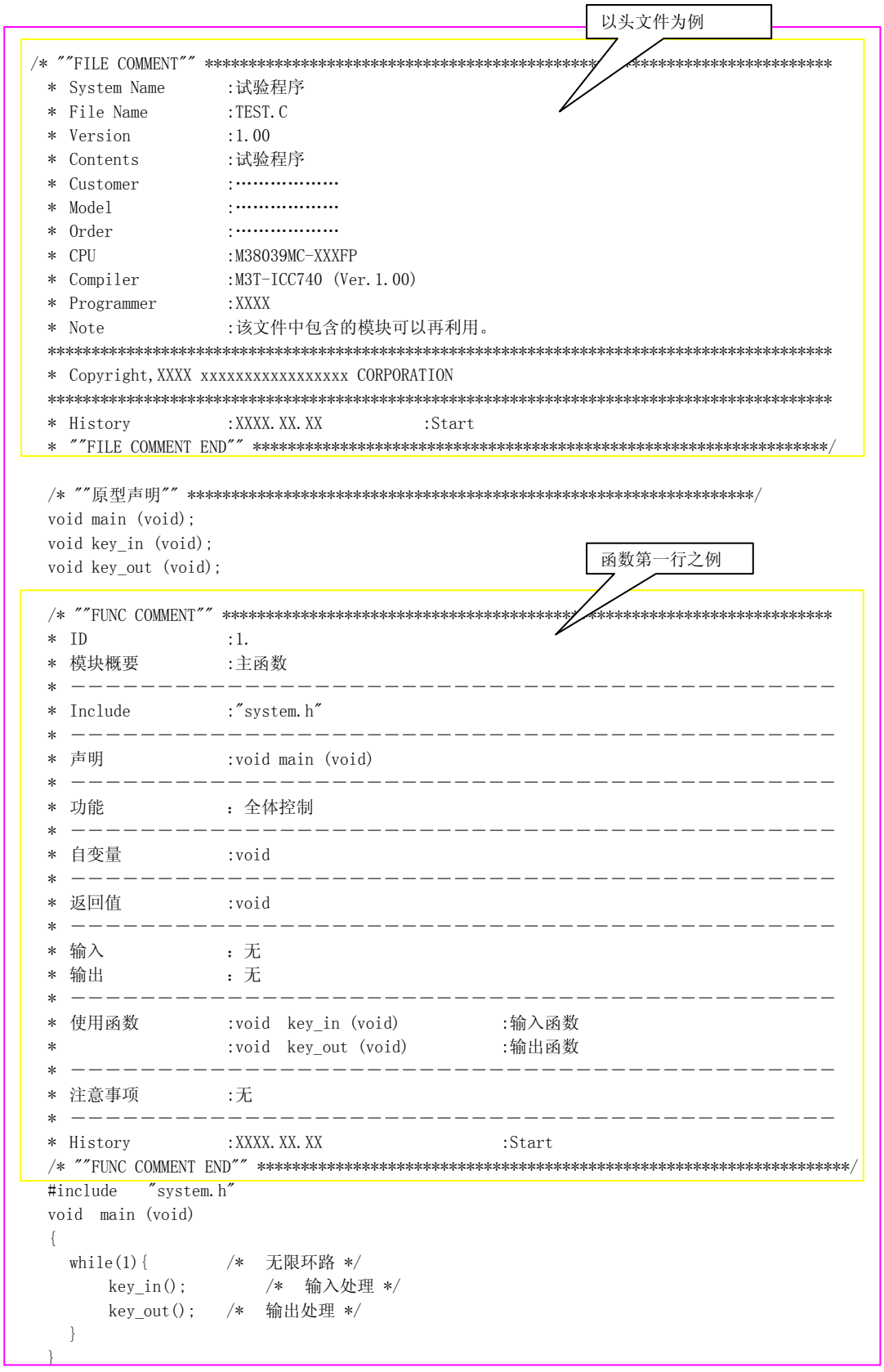
下面举例介绍用 C 语言编程的格式。

(1) 对每种功能建立一个函数。
(2) 每个函数不要太长，若非必要，不要多于 50 行。
(3) 在一行里不要写多个执行语句。
(4) 每一级处理，采用缩进格式（通常缩进 4 Tab）。
(5) 有效的加入必要的注释，说明程序的流程。
(6) 当使用多个源文件建立一个程序的情况下，把程序的公共部分独立出来，构成一个共享文件。



注释的描述方法

注释的描述方法也是程序易读的一个重要方面。函数的第一行,说明文件和函数的功能,程序的流程等。




```
/* ""FILE COMMENT"" ************************************************** ***********************
  * System Name       :试验程序
  * File Name         :TEST.C
  * Version           :1.00
  * Contents          :试验程序
  * Customer          :·····················
  * Model             :·····················
  * Order             :·····················
  * CPU               :M38039MC-XXXFP
  * Compiler          :M3T-ICC740 (Ver.1.00)
  * Programmer        :XXXX
  * Note              :该文件中包含的模块可以再利用。
  ************************************************************************************************
  * Copyright,XXXX xxxxxxxxxxxxxxxx CORPORATION
  ************************************************************************************************
  * History           :XXXX.XX.XX           :Start
  * ""FILE COMMENT END"" ********************************************************************/

  /* ""原型声明"" ****************************************************************/
  void main (void);
  void key_in (void);
  void key_out (void);

  /* ""FUNC COMMENT"" ************************************************ ***********************
  * ID                :1.
  * 模块概要          :主函数
  * ------------------------------------------------------------------
  * Include           :"system.h"
  * ------------------------------------------------------------------
  * 声明              :void main (void)
  * ------------------------------------------------------------------
  * 功能              : 全体控制
  * ------------------------------------------------------------------
  * 自变量            :void
  * ------------------------------------------------------------------
  * 返回值            :void
  * ------------------------------------------------------------------
  * 输入              : 无
  * 输出              : 无
  * ------------------------------------------------------------------
  * 使用函数          :void  key_in (void)          :输入函数
  *                   :void  key_out (void)         :输出函数
  * ------------------------------------------------------------------
  * 注意事项          :无
  * ------------------------------------------------------------------
  * History           :XXXX.XX.XX                   :Start
  /* ""FUNC COMMENT END"" ********************************************************************/
  #include    "system.h"
  void  main (void)
  {
    while(1){         /*  无限环路 */
        key_in();         /*  输入处理 */
        key_out();    /*  输出处理 */
    }
  }
```

专栏　ICC740 的保留字

下面是 ICC740 的保留字,不能用做变量名或函数名。

__asm ※	do	int	short	unsigned
auto	double	interrupt ※	signed	void
bit ※	else	long	sizeof	volatile
break	enum	monitor ※	static	while
case	extern	no_init ※	struct	zpage ※
char	float	npage ※	switch	
const	for	register	tiny_func ※	
continue	goto	return	typedef	
default	if	sfr ※	union	

※　「-e」操作使用时,成为保留字。

1.2 数据类型

1.2.1 C语言中使用的“常量”

C 语言中可使用整型常量、实型常量、单字符常量、字符串常量等 4 种常量。
下面介绍使用实型常量时的表示方法及注意点。

整型常量

整型常量能用十进制整数、十六进制整数以及八进制整数三种方法来描述,如下表所示。整型数据使用大写字母和小写字母均可。

种类	表示方法	例
十进制数	通常的数学表示(什么也不带)	127, +127, -56
十六进制数	数字之前加「0x」或「0X」	0x3b, 0x3B
八进制数	数字之前加「0」	07, 041

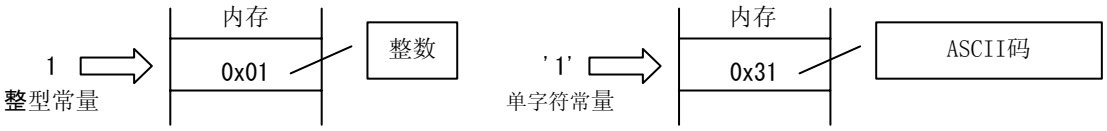
实型常量(浮点常量)

带符号的实数用十进制数表示,称为浮点常量。十进制数可以用通常的小数点和「e」或「E」的指数来表示。

① 通常的小数点表示　　例：175.5,-0.007
② 「e」或「E」的指数表示　　例：1.755e2,-7.0E-3

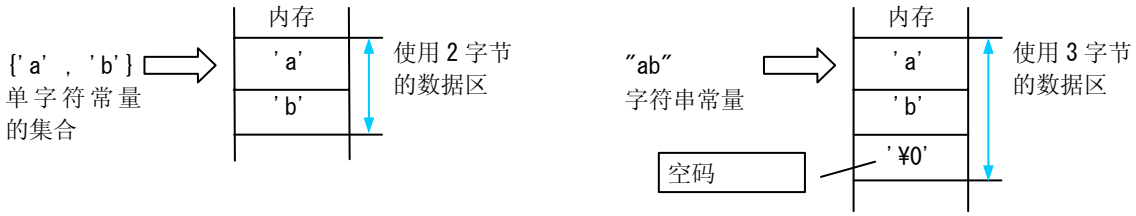
单字符常量

单字符常量用一对单引号（‘’)括起来。英文字母以外,控制码也能作为单字符常量使用。如下图所示,内部均使用 ASCII 码。



字符串常量

将英文字母或控制码用一对双引号（″）括起来，作为字符串常量来处理。
在字符串常量的最后面自动加空码‘\0’，表示字符串结束。
例：″abc″, ″012\n″, ″Hello!″



专栏：　控制码一览（Escape Sequence）

下面是 C 语言程序中经常使用的控制码（Escape Sequence）。

表示	内容
\f	改页（FF）
\n	改行复原（NL）
\r	回车（CR）
\t	水平 Tab（HT）
\\	\记号
\’	单引号
\″	双引号
\x 常数值	十六进制数
\常数值	八进制数
\0	空码

1.2.2 变量

C 语言程序中使用变量之前,必须对变量的数据类型进行声明。数据类型是由该变量上内存单元的多少和存储数值的大小决定的。

下面介绍 ICC740 中能处理变量的数据类型和声明方法。

ICC740 的基本数据类型

ICC740 能处理的数据类型。()内可以省略。

	数据类型	占用位数	数值范围
整数	char	8 位	0～255
	unsigned char		0～255
	signed char		-128～127
	unsigned short (int)	16 位	0～65535
	(signed) short (int)		-32768～32767
	unsigned int	16 位	0～65535
	(signed) int		-32768～32767
	unsigned long (int)	32 位	0～4294967295
	(signed) long (int)		-2147483648～2147483647
实数	float	32 位	有效位数 9 位
	double	32 位	有效位数 9 位
	long double	32 位	有效位数 9 位

※使用「-c」操作时,char 类型与 signed char 等价,可存放的数值范围也扩大到-128～127。

变量的声明与定义

变量的声明及定义以「数据类型　变量名;」的格式描述。

例：把变量 a 作为 char 类型来声明。

```
char  a;
```

使用「数据类型　变量名 ＝ 初始值;」的格式时，表示在进行定义的同时,可对该变量赋初始值。

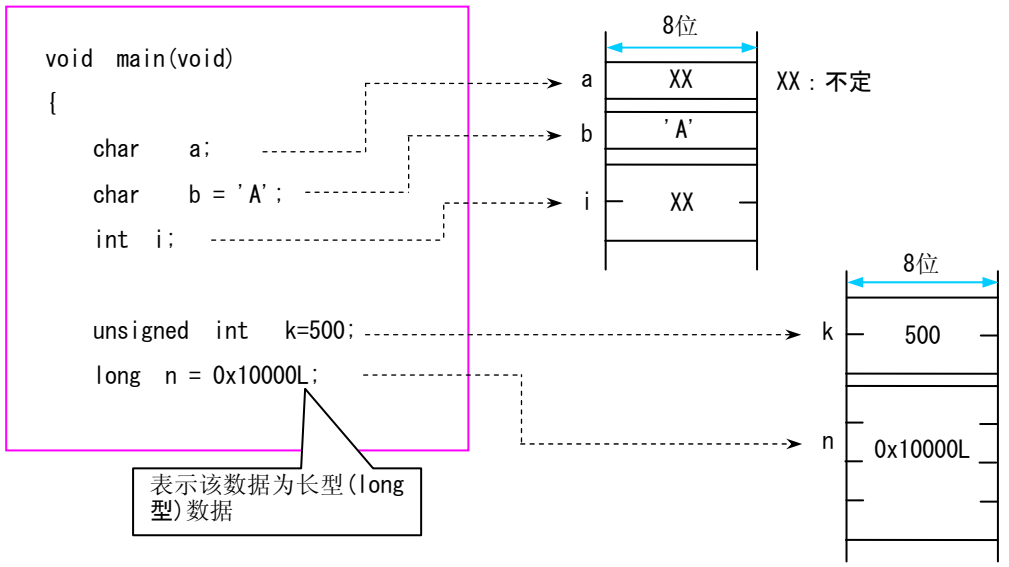
例：在 char 型的变量 a 中,设定'A'为初始值。

```
char  a = 'A';
```

同时声明/定义相同数据类型的几个变量时，变量名之间用逗号（','）分开。

例：int i,j;

例：int i=1, j=2;



1.2.3 类型限定(Qualifier)

声明变量或常量时,结合数据类型可描述该数据的特性。
下面介绍 ICC740 中处理数据的特性和指定方法。

有符号,或无符号要明确(signed/unsigned)

数据有符号时用"signed"表示,无符号时用"unsigned"表示。 声明时不描述该数据的类型时, 对 ICC740 而言, 只有 char 型数据无符号,其他的数据均为有符号的数据。

```
void  main(void)
{
    char    a;          与"unsigned char  a;"相同
    signed  char    s_a;

    int  b;             与"signed int  b;"相同
    unsigned  int   u_b;
              ┆
}
```

※ 使用「-c」操作时,char 型与 signed char 相同。

整数常数要明确声明(const)

在程序执行过程中数值完全不产生变化的数据声明时,用"const"描述。程序中如果存在使该常数产生变化的描述,ICC740 将输出错误信息。

```
void  main(void)
{
    char  a = 10;
    const signed  char  c_a=20;

    a = 5;
    c_a = 5;            发生错误!
}
```

禁止通过编译器进行最优化处理(volatile 修饰符)

ICC740 对程序处理过程中没有意义的指令进行最优化处理,避免形成不必要的指令码。但是,有些数据的变化与程序的处理无关,而随着中断的发生或电路板的输入而变化。对这种数据进行声明时,使用"volatile"来描述。对这种用类型修饰符描述的数据,ICC740 不进行最优化处理,直接输出命令码。

```
char port1;
char volatile port2;

void func(void)
{
        port1 = 0;
        port2 = 0;
        if( port1 == 0 ){
              ┆
        }
        if( port2 == 0 ){
              ┆
        }
}
```

声明数据时因没有使用"volatile"描述,在最优化处理中被除去,所以不输出指令码

声明数据时,因为使用了"volatile"来描述,所以不进行最优化处理而直接输出指令码

专栏:　声明文

对数据进行声明时,使用各种指定符或修饰符来描述。
以下为声明文。

声明指定符			声明符
存储类型指定符	类型修饰符	类型指定符	
static	const	char	数据名称
register	volatile	short	
auto		int	
extern		long	
typedef		float	
		struct	
		union	
		enum	
		void	
		signed	
		unsigned	

1.3 运算符

1.3.1 ICC740的运算符

ICC740 为编写程序规定了各种运算符。
以下,按不同用途分别介绍下述运算符(地址操作符和指针运算符除外)的描述方法和使用上的注意事项。

ICC740 中可以使用的运算符

ICC740 中可使用的运算符如下。

一目算术运算符	++ -- + -
二目算术运算符	+ - * / %
左移,右移运算符	<< >>
位运算符	& | ^ ~
关系运算符	> < >= <= == !=
逻辑运算符	&& || !
赋值运算符	= += -= *= /= %= <<= >>= &= |= ^=
条件运算符	?:
sizeof 运算符	sizeof
数据类型转换运算符	(型)
地址运算符	&
指针运算符	*
逗点运算符	,

1.3.2 数据运算符

数值计算所使用的主要的运算符包括用于进行计算的算术运算符和用于保存结果的赋值运算符。
下面介绍算术运算符和赋值运算符。

一目算术运算符

一目算术运算符指对一个变量返回一个结果。

运算符	描述形式	内容
++	++变量(前置式) 变量++(后置式)	自增运算
--	--变量(前置式) 变量--(后置式)	自减运算
+	+式	式中值返回
-	-式	式中值的符号相反,值返回

将自增运算符 (++)或自减运算符(--)与赋值运算符或关系运算符组合使用时,以前置式描述和以后置式描述的计算结果有时会不同。

<例>

前置式;增值或减值后赋值。

b = ++a; ⟶ a = a+1 ; b = a;

后置式;赋值后再进行增值或减值。

b = a++; ⟶ b = a ; a = a+1;

二目算术运算符

除了通常的四则运算之外,还可进行整数÷整数的「商」运算。

运算符	描述形式	内容
+	式 1 + 式 2	式 1 与式 2 的和返回
-	式 1 - 式 2	式 1 与式 2 的差返回
*	式 1 * 式 2	式 1 与式 2 的积返回
/	式 1 / 式 2	式 1 与式 2 的商返回
%	式 1 % 式 2	式 1 与式 2 商的余值返回

赋值运算符

「式 1＝式 2」表示将式 2 的值赋给式 1。而且,赋值运算符'='可以与前面介绍的算术运算符以及后面介绍的位运算符,移动运算符组合进行描述（「组合赋值运算符」）。在这种情况下,一定把赋值运算符'='在右侧描述。

运算符	描述形式	内容
=	式 1 = 式 2	将式 2 的值赋给式 1(代入式 1 中)
+=	式 1 += 式 2	将式 1 的值与式 2 的值相加后,赋给式 1
-=	式 1 -= 式 2	将式 1 的值与式 2 的值相减后,赋给式 1
*=	式 1 *= 式 2	将式 1 的值与式 2 的值相乘后,赋给式 1
/=	式 1 /= 式 2	将式 1 的值与式 2 的值相除后,赋给式 1
%=	式 1 %= 式 2	将式 1 的值与式 2 的值相除后的余数赋给式 1
<<=	式 1 <<= 式 2	将式 1 的值以式 2 的值的大小向左移动后,赋给式 1
>>=	式 1 >>= 式 2	将式 1 的式以式 2 的值的大小向右移动后,赋给式 1
&=	式 1 &= 式 2	将式 1 的值与式 2 的值的位逻辑运算“与”赋给式 1
|=	式 1 |= 式 2	将式 1 的值与式 2 的值的位逻辑运算“或”赋给式 1
^=	式 1 ^= 式 2	将式 1 的值与式 2 的值的“异或”赋给式 1

专栏　隐含类型转换

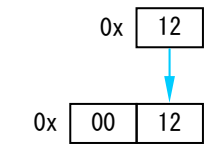
在 ICC740 中,不同类型的数据间进行运算时,遵循下述规则进行隐含类型转换。

- 字长按长数据类型。
- 赋值时,与左边的类型一致。

```
char    byte = 0x12;
int     word = 0x3456;
```

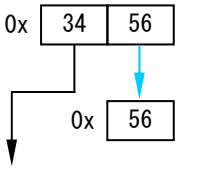
的情况下

```
word = byte;
       /* int ← char */
```



0x00 扩展

```
byte = word;
       /* char ← int */
```



删掉高位 1 字节

1.3.3 数据位运算符

数据加工时经常使用的运算符有位运算符和移位运算符。
下面介绍位运算符和移位运算符。

位运算符

使用位运算符时,可以进行数据的屏蔽处理和有效变换。

运算符	描述形式	内容
&	式 1 & 式 2	式 1 的值与式 2 的值按位与返回
|	式 1 | 式 2	式 1 的值与式 2 的值按位或返回
^	式 1 ^ 式 2	式 1 的值与式 2 的值按位异或返回
~	~ 式 1	式 1 的值按位取反后返回

移位运算符(shift operator)

不仅是进行移位,而且可以进行简单的乘除运算(请参考「使用移位运算符的乘除算」)。

运算符	描述形式	内容
<<	式 1 << 式 2	将式 1 的值按式 2 左移后,将结果返回
>>	式 1 >> 式 2	将式 1 的值按式 2 右移后,将结果返回

算术移位与逻辑移位的比较

进行右移时,对象数据有符号或无符号,对结果有影响。
- 无符号时→「逻辑移位」：最高位的位插入'0'。
- 有符号时→「算术移位」：保留符号进行移位。也就是说,正数时从最高位插入'0',负数时从最高位插入'1'。

	〈无符号〉 unsigned int i = 0xFC18 (i= 64520)	〈负数〉 signed int i = 0xFC18 (i= -1000)		〈正数〉 signed int i = 0x03E8 (i= +1000)	
	1111 1100 0001 1000	1111 1100 0001 1000		0000 0011 1110 1000	
i >> 1	0111 1110 0000 1100	1111 1110 0000 1100	(-500)	0000 0001 1111 0100	(+500)
i >> 2	0011 1111 0000 0110	1111 1111 0000 0110	(-250)	0000 0000 1111 1010	(+250)
i >> 3	0001 1111 1000 0011	1111 1111 1000 0011	(-125)	0000 0000 0111 1101	(+125)
	逻辑移位	算术移位 (保留符号)			

专栏　使用移位运算符的乘除法运算

使用移位运算符可简单地进行乘除计算。与使用通常的乘除运算符相比,运算速度加快。ICC740 中考虑这一点,对 "*2"、"*4"、"*8"等 不是生成乘法命令,而是生成移位命令。

- 乘法：进行移位运算。
  a*2 → a<<1
  a*4 → a<<2
  a*8 → a<<3

- 除法：通过检测出从下位被挤出的数据, 可以得到余数。
  a/4 → a>>2
  a/8 → a>>3
  a/16 → a>>4

1.3.4 条件运算符

控制语句中使用的条件运算符包括:关系运算符和逻辑运算符。两个运算符均在条件成立时返回’1’,条件不成立时返回’0’。

下面介绍关系运算符和逻辑运算符。

关系运算符

比较两个式子的大小关系。如果结果为真,返回’1’, 结果为假则返回’0’。

运算符	描述形式	内容
<	式 1 < 式 2	式 1 的值如果比式 2 的值小则为真,否则为假
<=	式 1 <= 式 2	式 1 的值如果小于或等于式 2 的值则为真, 否则为假
>	式 1 > 式 2	式 1 的值如果大于式 2 的值则为真, 否则为假
>=	式 1 >= 式 2	式 1 的值如果大于或等于式 2 的值则为真, 否则为假
==	式 1 == 式 2	式 1 的值如果等于式 2 的值则为真, 否则为假
!=	式 1 != 式 2	式 1 的值如果不等于式 2 的值则为真, 否则为假

逻辑运算符

与关系运算符同时使用, 可调查几个条件式的组合条件。

运算符	描述形式	内容
&&	式 1 && 式 2	式 1 与式 2 两者均为真的情况下真,否则为假
||	式 1 || 式 2	式 1 与式 2 两者均为假的情况下假,否则为真
!	!式	式中如果条件为真则变成假,如果条件为假则变成真

1.3.5 其他运算符

下面介绍 C 语言中具有特殊性质的 6 种运算符。

条件运算符

如果条件式为真则执行式 1,如果为假则执行式 2。在条件式、表达式 1 和式 2 的执行语句较短的情况下、如果使用条件运算符，则条件分支的编码可简单进行。下面介绍举例条件运算符。

运算符	描述形式	内容
? :	条件式 ? 式 1 : 式 2	条件式如果为真则执行式 1,如果为假则执行式 2

・选择值大的一个

```
c = a > b ? a : b ;
```

=

```
if(a > b){
        c = a ;
}
else{
        c = b ;
}
```

・求绝对值

```
c = a > 0 ? a : -a ;
```

=

```
if(a > 0){
        c = a ;
}
else{
        c = -a ;
}
```

长度运算符

需要了解某种数据类型或表达式所使用的内存字节数时，使用此运算符。

运算符	描述形式	内容
sizeof	sizeof 式 sizeof(数据类型)	表达式或数据类型的内存使用量以字节单位返回

强制类型转换运算符(cast operator)

不同类型的数据间进行运算时,运算中使用的数据向式中值最大的数据类型进行隐含类型转换。但是,有时会产生意想不到的问题,因此利用强制类型转换运算符明确类型转换。

运算符	描述形式	内容
(类型)	(新数据类型)变量	变量的数据类型转换为新型数据类型

地址运算符

地址运算符的作用是将分配给变量的存储单元的地址值返回。变量也可以是数组，在这种情况下，元素号码所显示的地址值被返回。

运算符	描述形式	内容
&	&变量	变量的地址值返回

指针运算符

指针变量指定存储单元的内容。

运算符	描述形式	内容
*	*变量	指针变量指定存储单元的内容

逗号运算符

从式 1 到式 2，从左边开始按顺序执行。几个短语句需要进行同时处理时使用。

运算符	描述形式	内容
,	式 1 ， 式 2	式 1，式 2 从左边开始按顺序执行

1.3.6 运算符的优先级

与算术运算符相同，C 语言中使用的运算符也有优先级与结合规则。
下面介绍运算符的优先级与结合规则。

优先级与结合规则

如果一个表达式中包含几个运算符时，首先从优先级高的内容开始计算。结合规则表示同时存在几个优先级的运算符时，从左边或右边的哪一边开始计算。

优先级	运算符的种类	运算符	结合规则
高	式	() [] -> . (注 1)	→
↑	一目算术运算符 etc	! ~ ++ -- + - * (注 2) & (注 3) (型) sizeof	←
	乘除法运算符	* (注 4) / %	→
	加减法运算符	+ -	→
	位运算符	<< >>	→
	关系运算符(比较)	< <= > >=	→
	关系运算符(等价)	== !=	→
	位运算符 (AND)	&	→
	位运算符 (EOR)	^	→
	位运算符 (OR)	|	→
	逻辑运算符 (AND)	&&	→
	逻辑运算符(OR)	||*	→
	条件运算符	?:	←
↓	赋值运算符	= += -= *= /= %= &= ^= |= <<= >>=	←
低	逗号运算符	,	→

(注 1) '.'是指定构造体与共用体成员的成员运算符。
(注 2) '*'是表示指针变量的指针运算符。
(注 3) '&'是表示变量地址的地址运算符。
(注 4) '*'是表示乘法运算的算术运算符。

1.3.7 易出错运算符的使用例

对运算符的隐含类型变换和优先级的解释如果发生错误,程序将不能正常运行。
下面介绍容易发生错误的运算符的使用例及相应的处理方法。

隐含类型变换的解释错误及处理方法

不同类型的数据间进行运算时,进行下述「隐含类型转换」。
(A) 位长取长型数据的类型进行运算。
(B) 比 int 类型短的数据类型用于算术运算时,全部扩展为 int 类型。
(C) 常数按 int 类型处理。

(1)

```
unsigned char a,b;
a = 0;
b = 5;
if( unsigned char (a - 1) >= b ){
        ¦
}
else{
        ¦
}
```

因为表达式(a-1)使用数据转换类型运算,
左边为 unsigned 0x00 - unsigned 0x01 = unsigned 0xFF,
也就是 255。
因 255 >= 5 所以被判断为真。

下面介绍 if 语句为假,程序不能按预定正常运行的情况。

(2)

```
unsigned char a,b;
a = 0;
b = 5;
if( (a - 1) >= b){
        ¦
}
else{
        ¦
}
```

表达式(a-1)变为 unsigned char - signed int,通过「隐含类型转换」,
将 unsigned char 类型转换为 signed int 类型,
左边为 signed 0x0000 − signed 0x0001 = signed 0x0000 + signed 0xFFFF
也就是-1。因为-1<=5, 判断为假。

(3)

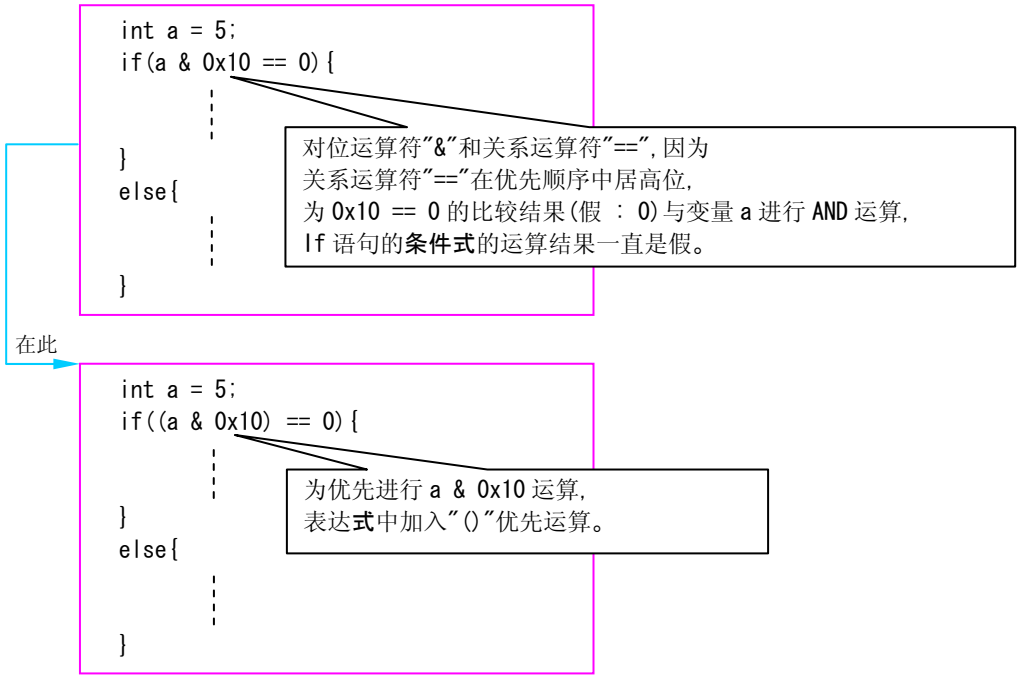
```
unsigned char a,b;
a = 0;
b = 5;
if( ( a - (unsigned char)1 ) >= b ){
        ¦
}
else{
        ¦
}
```

因为将常数 1 转换成 unsigned char 类型,
表达式左边成为 unsigned char - unsigned char,
通过(B) int 扩张为 signed int 类型, 与(2)的运算相同。
因为-1 >= 5, 所以判断为假。

另外,最后请确认列表文件(list file)和汇编文件(assembly file)。根据记述方法的不同,会产生程序大小变化,以及调用随机库等现象。

对优先级的解释发生错误时的处理方法

一个表达式中包含几个运算符的情况下，需要说明运算符的优先级和结合规则。而且，为使程序按设计运行，在表达式中加入"()"来处理。



1.4 控制语句

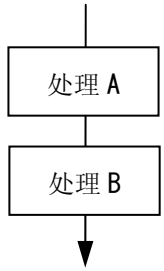
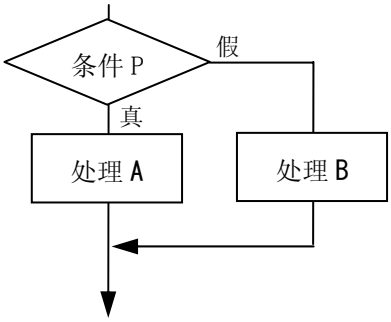
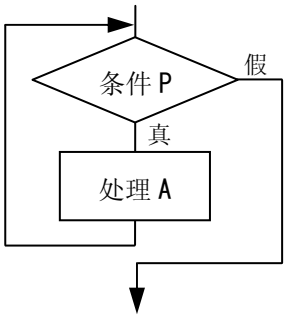
1.4.1 程序的结构化

C 言語中结构化程序包括:顺序处理,分支结构,循环结构几个部分,均可用控制语句来描述。因此,用 C 言語描述的结构化程序,其程序流程易懂。

下面举例介绍控制语句的描述方法。

程序的构造化

程序易懂的最关键一点在于程序流程是否易读。程序的流程不能任意定,在遵循顺序结构,分支结构,循环结构这三个规则下进行设计,这种方法被称为结构化编程。结构化编程用下面三个基本结构表示。

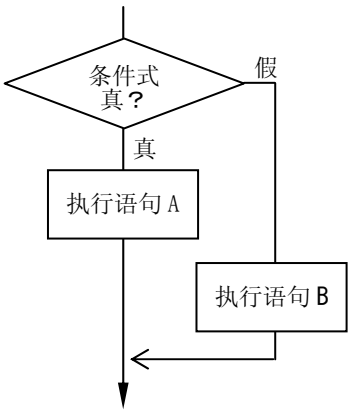
顺序结构		从上向下执行。
分支结构		根据条件 P 的真与假 分别进行 A 处理和 B 处理
循环结构		条件 P 成立期间, A 处理循环进行。

1.4.2 分支结构

描述分支结构的控制语句包括 if-else 语句, else-if 语句, switch-case 语句。
下面介绍这些控制语句的描述方法与示例说明。

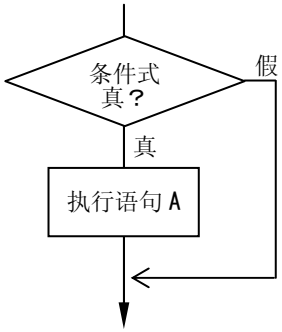
if-else 语句

如果条件是真的情况下,立即执行下一个程序块,反之如果是假,则执行 else 程序块。
"else"程序块可以省略。




```
if(条件式){
        执行语句 A
}
else{
        执行语句 B
}
```

・省略 else 语句的情况下




```
if(条件式){
    执行语句 A
}
```

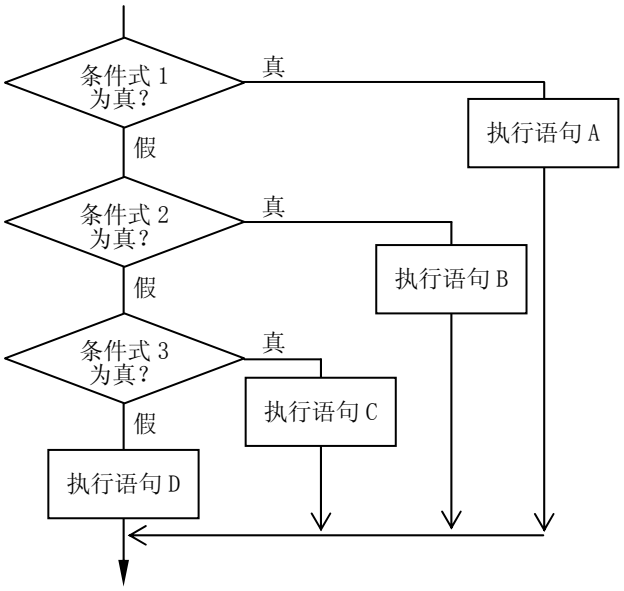
计数器(if-else 语句的描述例)

对秒"second"与分"minute"进行累计计数。如果每隔一秒调用模块一次,
可以起到计时的作用。

```
void count_up(void);                    "count_up"函数声明(参照「1.5  函数」)
unsigned int second = 0;                "second"(秒计时)与
unsigned int minute = 0;                "minute"(分计时)用变量声明

void count_up(void)                     "count_up"函数的定义
{
    if(second >= 59){                   如果在 59 秒以上
        second = 0;                         「秒」重置
        minute ++;                          「分」累加计数
    }
    else{                               如果不满 59 秒
        second ++;                          「秒」累加计数
    }
}
```

else-if 语句的嵌套

对于几个条件,需要分开进行 3 个以上处理时使用该语句。各条件为真情况下想要进行的处理马上在下一个程序块进行描述。所有的条件都不符合情况下,其处理方式在最后的"else"程序块描述。




```
if(条件式 1){
        执行语句 A
}
else if(条件式 2){
        执行语句 B
}
else if(条件式 3){
        执行语句 C
}
else{
        执行语句 D
}
```

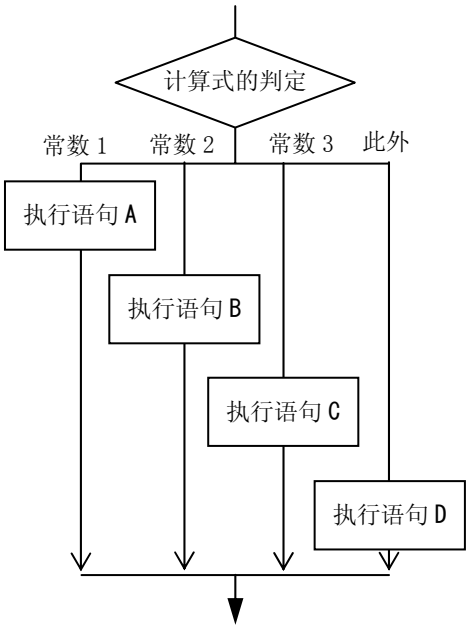
四则运算的切换-1-(else-if 语句的描述例)

根据输入数据"sw"的内容切换运算。

```
void select(void);          ← "select"函数声明(参照「1.5  函数」)

int a = 29, b = 40;         ← 使用的变量声明
long int ans;
char sw;

void select(void)           ← "select"函数的定义
{
    if(sw == 0){            ← "sw"的内容如果是 0
        ans = a + b;            进行加法运算
    }
    else if(sw == 1){       ← "sw"的内容如果是 1
        ans = a - b;            进行减法运算
    }
    else if(sw == 2){       ← "sw"的内容如果是 2
        ans = a * b;            进行乘法运算
    }
    else if(sw == 3){       ← "sw"的内容如果是 3
        ans = a / b;            进行除法运算
    }
    else{                   ← "sw"的内容如果是 4 以上
        error();                进行错误处理
    }
}
```

switch-case 语句

根据计算式的结果需要进行几个选择处理时使用该语句。因为计算式的结果为常数,所以不能使用关系运算符。




```
switch(式){

        case    常数 1  :  执行语句 A
                           break;
        case    常数 2  :   执行语句 B
                           break;
        case    常数 3  :   执行语句 C
                           break;
              default   :   执行语句 D
                           break;
}
```

四则运算的切换-2-(switch-case 语句的描述例)

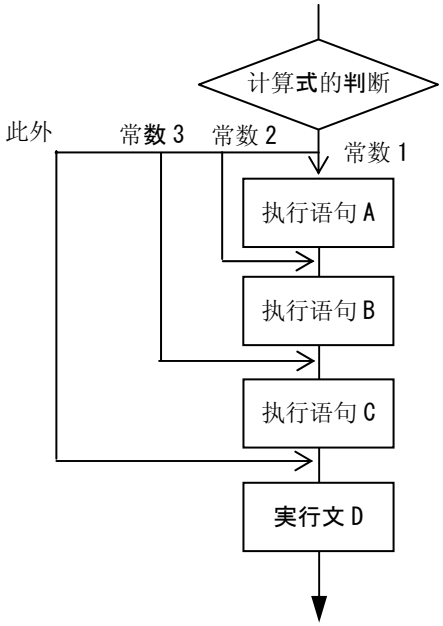
根据输入数据"sw"的内容来切换运算。

```
void select(void);                  ← "select"函数声明(参照「1.5  函数」)

int a = 29, b = 40;                 ← 使用变量的声明
long int ans;
char sw;

void select(void)                   ← "select"函数的定义
{
    switch(sw){                     ← "sw"的内容判断

        case 0 : ans = a + b;       ← "sw"的内容如果是 0 进行加法运算
                 break;
        case 1 : ans = a - b;       ← "sw"的内容如果是 1 进行减法运算
                 break;
        case 2 : ans = a * b;       ← "sw"的内容如果是 2 进行乘法运算
                 break;
        case 3 : ans = a / b;       ← "sw"的内容如果是 3 进行除法运算
                 break;
        default: error();           ← "sw"的内容如果大于 4 进行错误处理
                 break;
    }
}
```

专栏　没有 break 的 switch-case 语句

switch-case 各执行语句通常以 break 语句来结束。

如果程序块的最后没有 break 语句，该程序块一旦结束，按上下顺序执行下一个程序块。也就是说，根据算式的值可以改变运行处理的开始位置。



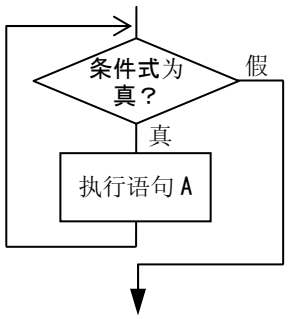

```
switch(式){

        case    常数 1  :   执行语句 A

        case    常数 2  :   执行语句 B

        case    常数 3  :   执行语句 C

                default  :   执行语句 D

}
```

1.4.3 循环结构

描述循环结构的控制语句有 while 语句,for 语句和 do-while 语句。
下面举例介绍控制语句的描述方法。

While 语句

在条件式成立期间,循环进行程序块内部的处理。条件式中描述 0 以外的常数时,条件式则一直保持为真,可以实现无限循环。




```
while(条件式){
        执行语句 A
}
```

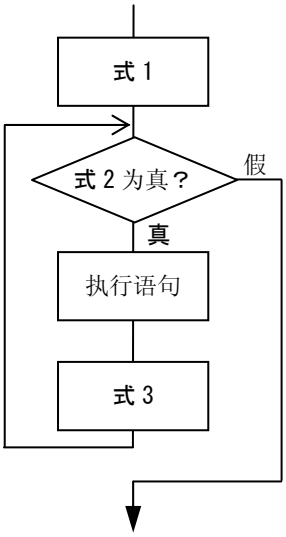
求总和-1-(while 语句的描述例)

求从 1 到 100 为止的整数和。

```
void sum(void);                  ←  "sum"函数声明(参照「1.5  函数」)
unsigned int total = 0;          ←  使用的变量声明
void sum(void)                   ←  "sum"函数的定义
{
    unsigned int i = 1;          ←  计数用的变量宣言,初始化
    while(i <= 100){             ←  计数的内容达到 100 为止进行循环
        total += i;
        i++;                     ←  令计数的内容变化
    }
}
```

For 语句

正如 while 语句例子所示,使用计数器进行循环处理时,在判断条件的同时,必须进行计数器初始化以及计数内容变化处理。在 For 语句中,这些处理可以与条件式同时描述。初始化(式 1),条件式(式 2),处理(式 3)可分别省略。但是,不管省略哪个计算式,请一定插入式间的分号';'。另外,for 语句内容可以被前面讲述的 while 语句改写。




```
for(式 1 ; 式 2 ; 式 3){

        执行语句

}
```

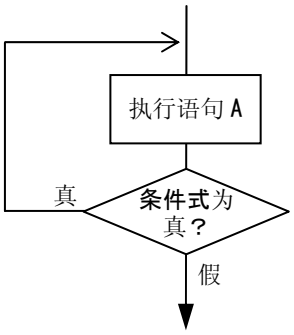
求总和-2-(for 语句的描述例)

求从 1 到 100 的整数和。

```
void sum(void);                          "sum"函数声明(请参考「1.5  函数」)
unsigned int total = 0;                  所使用的变量声明
void sum(void)                           "sum"函数的定义
{
    unsigned int i = 1;                  用于计数器的变量的声明

    for(i = 1; i <= 100; i++){           计数器的内容从 1 开始到 100 为止进行循环
        total += i;
    }
}
```

do-while 语句

与 for 语句及 while 语句不同,处理执行后进行条件判断(「后判定」)。在 for 语句及 while 语句中,根据条件不同,有些处理可能一次也没有执行过,而在 do-while 语句中必须然进行一次处理。




```
do{
        执行语句
}while(条件式);
```

求总和-3-(do-while 语句的描述例)

求从 1 到 100 为止的整数和。

```
void sum(void);                  ← "sum"函数声明(参照「1.5  函数」)
unsigned int total = 0;          ← 使用的变量声明
void sum(void)                   ← "sum"函数的定义
{
    unsigned int i = 1;          ← 计数器用变量的声明, 初始化

    do{
        i ++;
        total += i;
    }while(i < 100);             ← 计数器的内容达到 100 为止进行循环
}
```

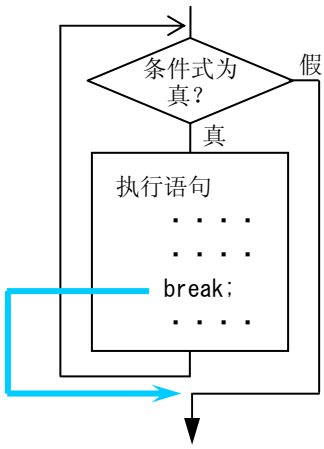
1.4.4 中断结构

用于中断结构的跳出控制语句(辅助控制语句)包括 break 语句, continue 语句, goto 语句。
下面介绍上述控制语句的描述方法与操作说明。

Break 语句

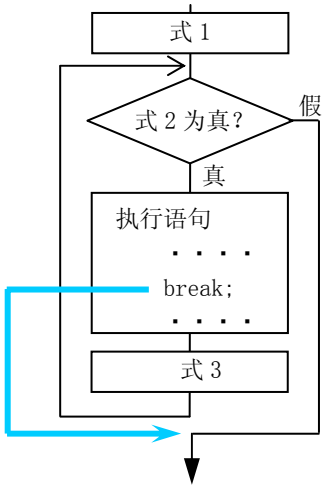
该语句用于反复处理及 switch-case 语句中。执行"break;"时处理中断,只有 1 个程序块跳出。

・使用 while 语句的情况下




```
while(条件式){
        ・・・・
        ・・・・
        break;
        ・・・・
}

for(式 1 ; 式 2 ; 式 3){
        ・・・・
        break;
        ・・・・
}
```

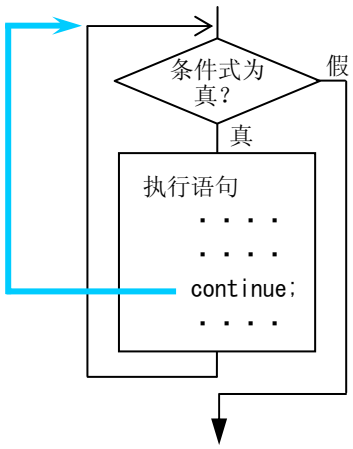
・使用 for 语句的情况下



Continue 语句

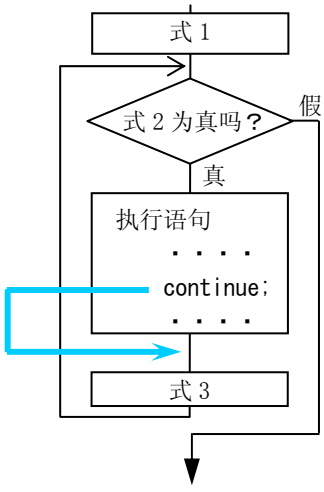
该语句在循环处理中使用。执行"continue;"时处理中断。while 语句在中断后返回到条件判断,for 语句中在式 3 执行后,返回到条件判断。

・使用 while 语句的情况下




```
while(条件式){
        ・・・・
        ・・・・
        continue;
        ・・・・
}

for(式 1 ; 式 2 ; 式 3){
        ・・・・
        continue;
        ・・・・
}
```

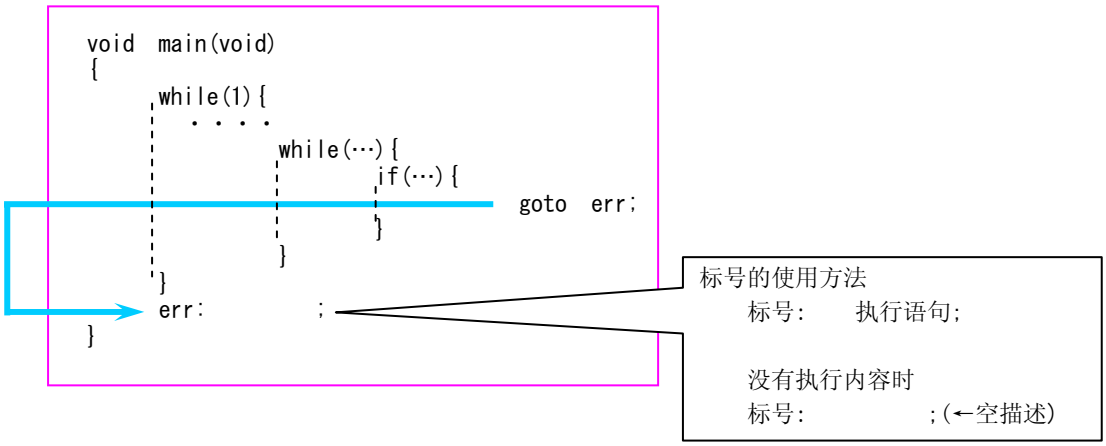
・使用 for 语句的情况下



Goto 语句

执行 Goto 语句时,程序可无条件的转向 goto 语句后标号所在的位置,执行其后面的语句。与 break 语句及 continue 语句不同,在函数内的任何位置都可以跳跃,可以一次跳过几个程序块而执行 goto 语句后标号所指程序。但是,因为与结构化程序的规则有不一致之处,因此,除错误处理等紧急必要的情况下以外,不推荐使用。

而且,选择的标号后必须有执行语句。没有执行内容的情况下必须有空百描述(只有';')。



1.5 函数

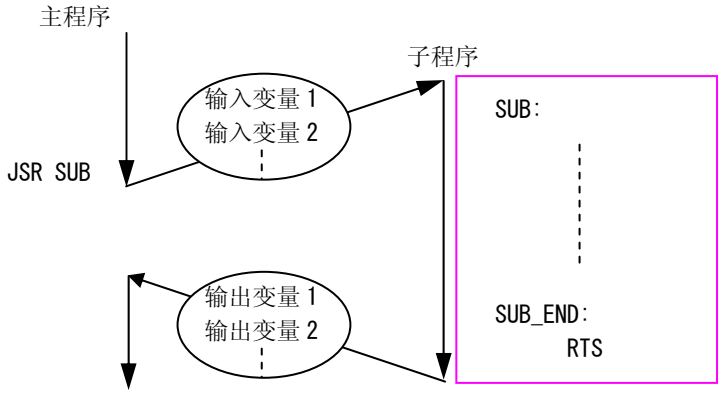
1.5.1 函数与子程序

正如子程序是构成汇编语言的基本单位一样，函数是构成 C 语言程序的基本单位。
下面介绍 ICC740 中函数的描述方法。

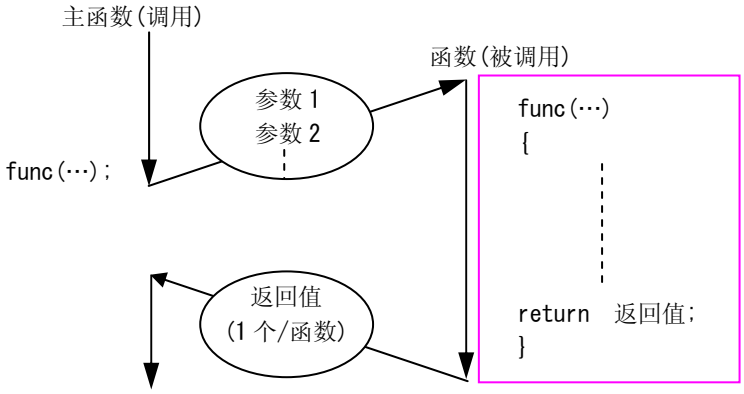
参数与返回值

函数间的数据的传递，由相当于子程序中输入变量的参数和相当于输出变量的返回值来进行。
汇编语言中对输入变量和输出变量的数量没有制约。但是，C 语言中规定，一个函数只能接收一个返回值，使用 return 语句返回。对于参数而言，一个函数的参数范围合计不能超过 256 字节，一旦超过 256 字节，编译器将发生错误。

・汇编语言的子程序



・C 语言的「函数」



1.5.2 函数的创建

使用函数时需要函数的原型声明,函数的定义和调用函数三个步骤。
下面说明各步骤的具体内容。

函数的原型声明

在 C 语言中,使用函数之前要进行函数的原型声明。
下面是函数原型声明的书写格式。

```
返回值的数据类型　　函数名(参数类型, ・・・・・・・・・・);
```

没有返回值或参数时,使用"void"形表达式来表示内容为空白。

函数的定义

在函数体中,为接收参数,需要定义「形式参数」的数据类型与名称。返回值使用 return 语句进行返回。
下面是函数定义的书写格式。

```
返回值的数据类型　　函数名(形参 1 的数据类型　形参 1, ・・・・・・・・・・);
{
        ┆
    return  返回值;
}
```

调用函数

调用函数时,为被调用的函数定义实参。使用赋值运算符接收被调用函数的返回值。

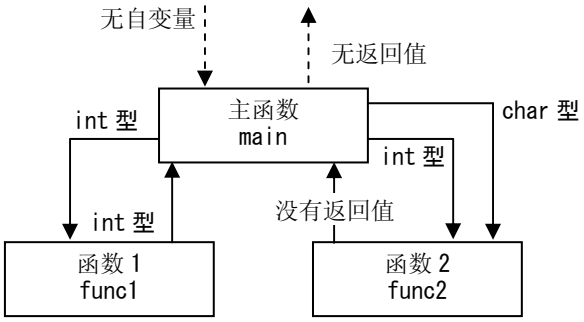
```
函数名(实参 1, ・・・・・・・・・・);
```

有返回值时

```
自量＝函数名(实参 1, ・・・・・・・・・・);
```

函数的描述例

在下面的例子中,介绍有相互关联的三个函数。




```
／＊　原型声明　＊／
void  main(void);
int  func1(int);
void  func2(int, char);

／＊　主函数　＊／
void  main()
{
    int  a = 40,b = 29;
    int  ans;
    char  c = 0xFF;

    ans = func1(a);
    func2(b, c);
}

／＊　函数 1 的定义　＊／
int  func1(int  x)
{
    int  z;
    z = x + 1;
    return  z;
}
／＊　函数2的定义　＊／
void  func2(int  y, char  m)
{
    :
}
```

以 a 作参数调用函数 1("func1")
返回值赋值给"ans"

以 b,c 作参数调用函数 2("func2")
无返回值

用 return 语句将返回值返回

1.5.3 函数间数据的传递

在 C 语言中，参数与返回值的传递是通过将各变量的值复制后传送的。因此，调用函数时使用的参数名称与被调用函数接受的参数(形参)名称没有必要保持一致。

在被调用函数内部进行内部处理时使用复制值(形参)，所以调用函数的变量本体不会被破坏。

由于上述原因，C 语言的函数独立性高，函数的再利用也比较容易。

下面说明一下函数间数据的传递。

求和(函数的描述例)

-32768～32767 范围内的任意两个整数作为参数组成求和的加法函数"add"，从主函数调用。

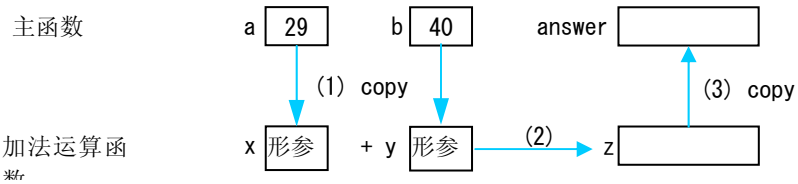
```
／＊　原型声明　＊／
void  main(void);
long  add(int, int);

／＊　主函数　＊／
void  main()
{
    long int  answer;
    int  a = 29, b = 40;

    answer = add(a, b);
}

／＊　加法运算函数　＊／
long  add(int  x, int  y)
{
    long int  z;

    z = (long int)x + y;
    return  z;
}
```

(1) 调用加法函数

(2) 加法运算

(3) 返回值返回

＜数据的流程＞



1.6 变量的存储类型

1.6.1 变量与函数的作用域（Scope）

有些变量和函数在整个程序内有效，而有些仅限于一个函数内部。根据各自性质不同其有效范围可以改变。变量及函数的有效范围被称为作用域(scope)。

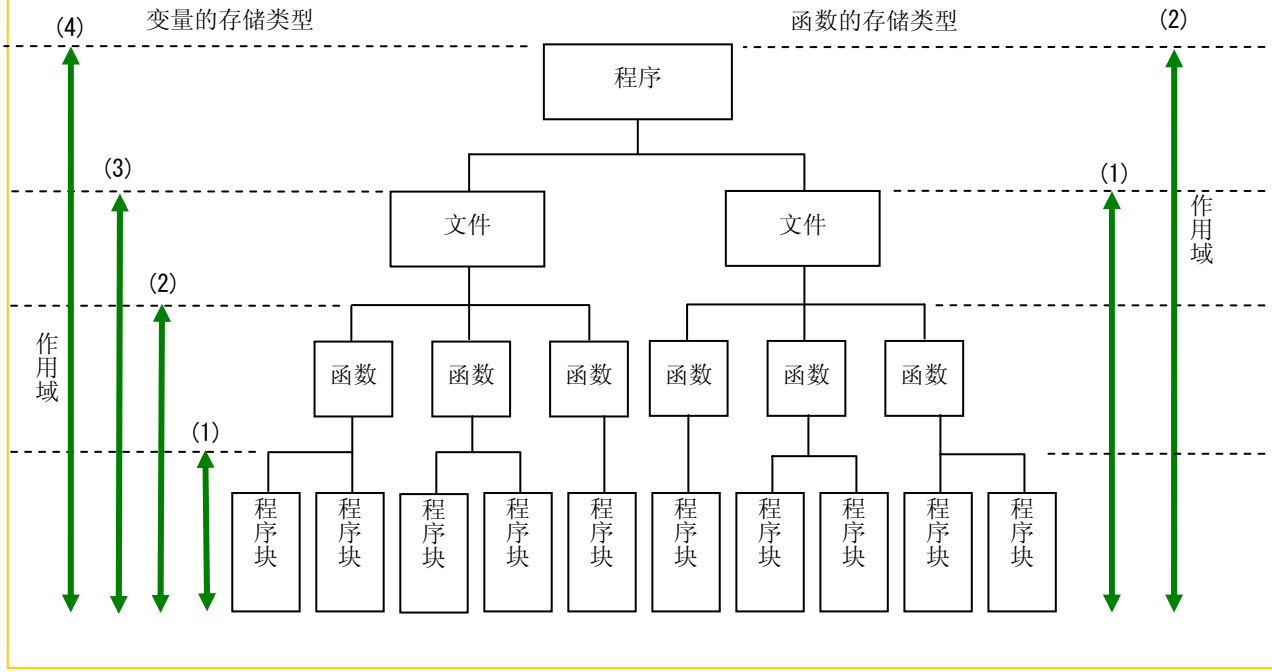
下面介绍变量与函数的存储类型及指定方法。

变量与函数的作用域

C 语言程序由数个源文件构成。更进一步，一个源文件由数个函数构成。也就是说，C 语言程序中的变量和函数由下述几种类型构成。

变量的作用域有下述 4 种类型。
(1) 仅在程序块内有效
(2) 仅在函数内有效
(3) 仅在文件内有效
(4) 在程序整体有效
函数的作用域有下述 2 种类型。
(1) 仅在文件内有效
(2) 在整个程序内有效

在 C 语言中可分别指定变量和函数的作用域。而且通过有效利用该作用域，对自定义变量和函数加保护（非公开）。反之，也可以公开共享。

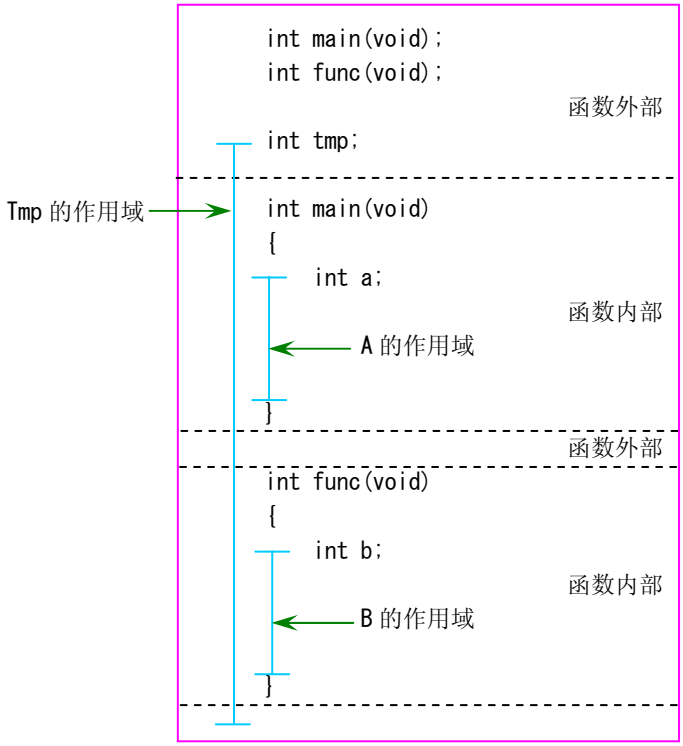


1.6.2 变量的存储类型

变量的存储类型在进行类型声明时指定,指定时要点如下:
(1)外部变量与内部变量(→进行类型声明的位置)
(2)存储类型说明符(→类型声明中附加指定符)
下面介绍变量存储类型的指定方法。

外部变量与内部变量

区分变量作用域的最简单方法，是根据进行变量类型声明的位置来决定。在函数外部声明的变量称为外部变量,函数内部声明的变量称为内部变量。外部变量是在声明以后任一函数均可引用的全局变量(global variable)。内部变量则是在声明后仅在该函数内部有效的局部变量(Local variable)。



存储类型说明符

变量的存储类型说明符包括 auto，static，register，extern 等。其表达式如下所示。

存储类型说明符　数据类型　变量名;

外部变量的存储类型

声明外部变量时,如果不指定存储类型说明符, 则该变量为静态 (全局) 变量,其作用域是整个程序有效。声明时用"static"来描述(即静态变量)时,变量的作用域只有在所声明的文件内有效。

使用在别的文件中所声明的外部变量时使用"extern"来描述, 比如源文件 2 的"mode"。

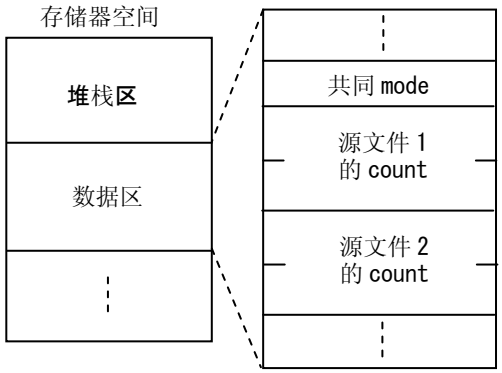
外部变量,不设定初始值的变量被分配在数据域中的 N_UDATA 段, 设定初始值的变量被分配在数据域上的 N_IDATA 段。

源文件 1

```
char mode;
static int count;

void func1(void)
{
    mode = STOP;
    count = 0;
        ¦
```

源文件 2

```
extern char mode;
static int count;

void func2(void)
{
    mode = BACK;
    count = 100;
        ¦
```

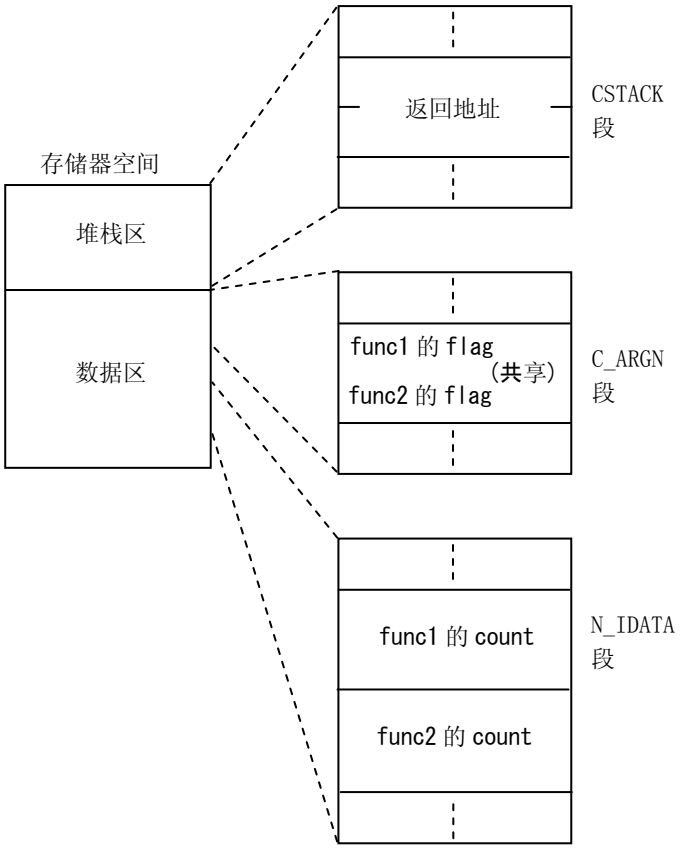


内部变量的存储类型

不带存储类型说明符所声明的内部变量,被分配在数据区域的 C_ARGN 段,该段由各函数共享。该函数每次被调用时都要进行初始化。声明时用"static"描述的内部变量。没有设定初始值的变量按 0 进行初始化后被分配在数据区域上的 N_UDATA 段,设定了初始值的变量按设定值进行初始化后被分配在数据领域上的 N_IDATA 段。只有在程序启动时进行初始化。

源文件

```
void func1(void)
{
    char flag = 0;
    static int count = 0;
        ¦
    flag = SET;
    count = count + 1;
    func2();
        ¦
}
void func2(void)
{
    char flag = 0;
    static int count = 0;
        ¦
    flag = SET;
    count = count + 1;
        ¦
}
        ¦
```

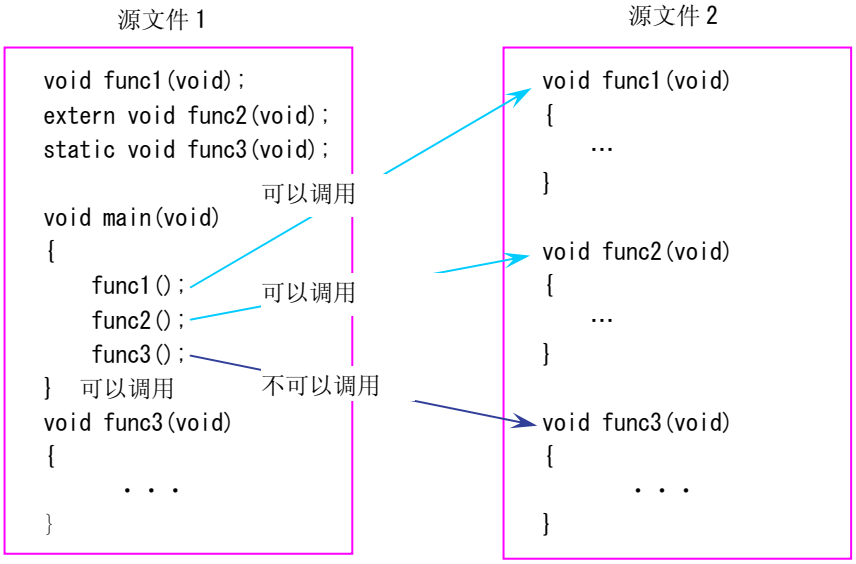


1.6.3 函数的存储类型

函数的存储类型由定义函数的一方与声明函数的一方双方指定。可以使用存储类型说明符「static」与「extern」。下面介绍函数存储类型的指定方法。

外部函数与内部函数

(1)定义函数时不指定存储类型的情况下
函数可以被其它源文件所调用,属外部函数。
(2)在函数原型中声明为 extern 类型的情况下
表明该函数不是在描述的源文件中所定义, 而是从其它的源文件调用的。但是,如果原型声明不使用"extern"描述, 函数不在源文件内部时,自动按照与用"extern"描述的进行相同处理。
(3)定义函数时声明为 static 类型的情况下
该函数不可以从其它源文件调用,属内部函数(静态函数)。



存储类型小结

变量的存储类型与函数的存储类型总结如下。

变量的存储类型

存储类型	外部变量	内部变量
省略存储类型说明符	从其它的源文件也可以调用的全局变量 [分配到 N_UDATA、N_IDATA 段] [数据保持 ○]	只有在函数内部有效的变量 [执行时分配到 C_ARGN 段] [数据保持 ×]
auto		只有在函数内部有效的变量 [执行时分配到 C_ARGN 段] [数据保持 ×]
static	从其它的源文件不能调用的局部变量 [分配到 N_UDATA, N_IDATA 段] [数据保持 ○]	只有在函数内部有效的变量 [分配到 N_UDATA、N_IDATA 段] [数据保持 ○]
register		只有在函数内部有效的变量 [执行时分配到 C_ARGN 段] [数据保持 ×]
extern	该变量是从其它源文件中调用的变量 [不分配]	该变量是从其它源文件中的调用的变量 (其它参数不能调用) [不分配]

函数的存储类型

存储类型	函数的种类
省略存储类型说明符	其它的源文件也可以调用执行的外部函数 [定义时指定]
static	其它的源文件不能调用的内部函数 [定义时指定]
extern	其它源文件中的函数 [由声明一方指定]

1.7 数组与指针

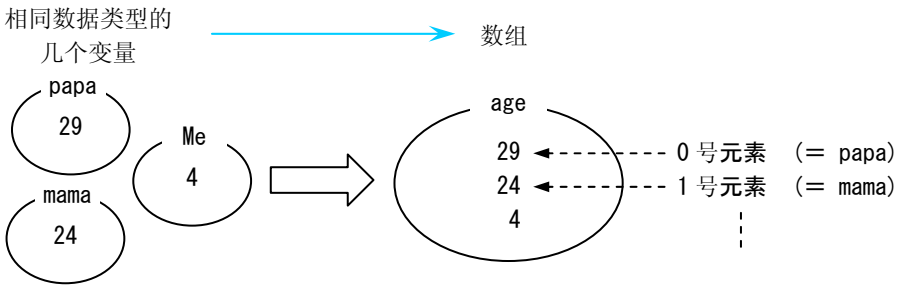
1.7.1 数组

在这一节中介绍数组的基本概念。

数组是什么?

下面以求全家人年龄之和的程序为例进行说明。家庭成员有父母(父亲＝29 岁、母＝24 岁)以及一个小孩(我＝4 岁)。

该程序中,随着家庭成员的增加变量也增加。C 语言中为处理这种情况,导入了数组这一概念。数组是指拥有相同类型(int 类型) 的数据集合体。在这个例子中,父亲的年龄(papa),母亲的年龄(mama)‥‥不作为单独的变量处理,而是将家庭成员的年龄作为一个集合体(age)来考虑。各数据成为集合体的元素。也就是说,0 号元素是父亲,1 号元素是母亲,2 号元素是我。



求家庭成员年龄之和-1-

求家庭成员(父、母、我)的年龄和。

```
void main(void)
{
    int papa = 29;
    int mama = 24;
    int boku = 4;
    int gokei;

    gokei = papa + mama + boku;
}
```

随着家庭成员的增加,变量类型声明, 初始化执行语句也增加

```
void main(void)
{
    int papa = 29;
    int mama = 24;
    int boku = 4;
    int imouto1 = 1;
    int imouto2 = 1;
        :
    int gokei;

    gokei = papa + mama + boku + imouto1 + imouto2 + …;
}
```

1.7.2 数组的使用

C 语言中使用的数组包括「一维数组」和「二维数组」。
下面介绍各数组的生成与引用方法。

一维数组

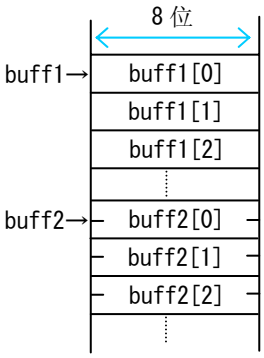
一维数组是一组有序数据的集合。一维数组的声明格式如下所示。

数据类型　数组名［元素个数］

进行上述声明后，该数组则被分配与元素个数相当的内存，数组名作为开头标号。
为引用一维数组，数组名称后加元素号码来表示。但是，因为元素号码从'0'开始，最后的元素号码则为元素数-1。

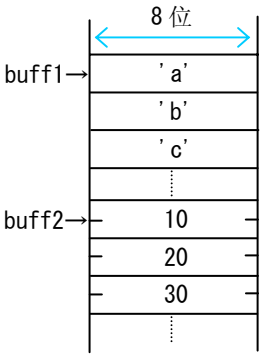
・1 维数组的声明

```
char buff1[3];
int buff2[3];
```



・一维数组的声明与初始化

```
char buff1[3] = {
  'a', 'b', 'c'
};

int buff2[3] = {
  10, 20, 30
};
```



求家族成员年龄之和-2-

利用数组求家庭成员年龄之和。

```
#define MAX 3 (注)

void main(void)
{
    int age[MAX];
    int gokei = 0;
    int i;

    age[0] = 29;
    age[1] = 24;
    age[2] = 4;

    for(i = 0; i < MAX; i++){
        gokei += age[i];
    }
}
```

或

```
#define MAX 3

void main(void)
{
    int age[MAX] = {
        29, 24, 4
    };

    int gokei = 0;
    int i;

    for(i = 0; i < MAX; i++){
        gokei += age[i];
    }
}
```

声明的同时进行初始化

使用数组时，可以利用反复语句，将元素个数设为变量.

(注)　#define MAX 3　：　MAX = 3 宏定义
（参照「1.9　预处理命令」）

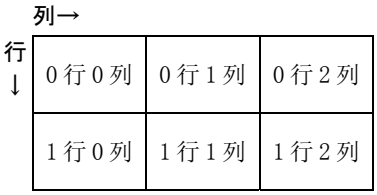
二维数组

二维数组是由行与列构成的平面二维数据集合。而且,二维数组也可以简化为一维数组。

```
数据类型　数组名［行数］［列数］;
```

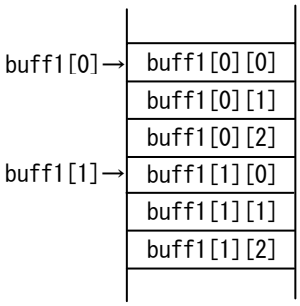
引用二维数组时,配列名前加行号码和列号码。行号码与列号码均从'0'开始,最后的号码为行数(列数)-1。

・二维数组的概念



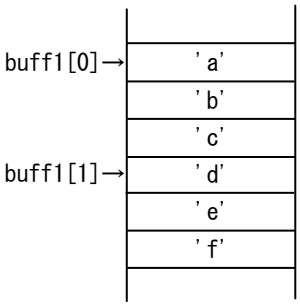
・二维数组的声明

```
char buff1[2][3];
```

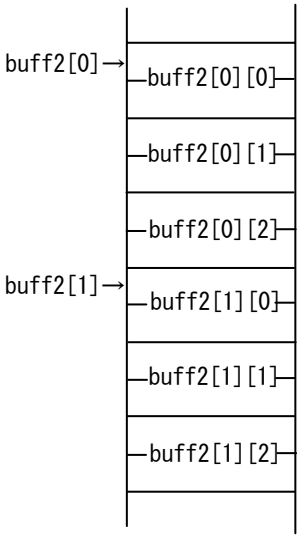


・二维数组的声明与初始化

```
char buff1[2][3] = {
  {'a', 'b', 'c'},
  {'d', 'e', 'f'}
};
```

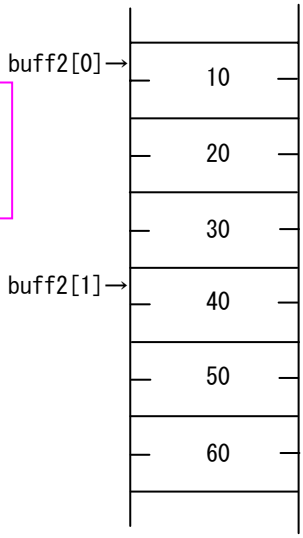



```
int buff2[2][3];
```




```
int buff2[][3] = {
  10, 20, 30, 40, 50, 60
};
```

声明的同时进行初始化时,
可省略行数的指定
(列数不能省略)



1.7.3 指针

指针是指向数据的变量,它表示一个地址。指针是处理数组的便利手段。
下面介绍的指针变量是把存储数据的地址作为变量来处理。相当于汇编语言中的间接寻址。
下面介绍指针变量的声明与引用方法。

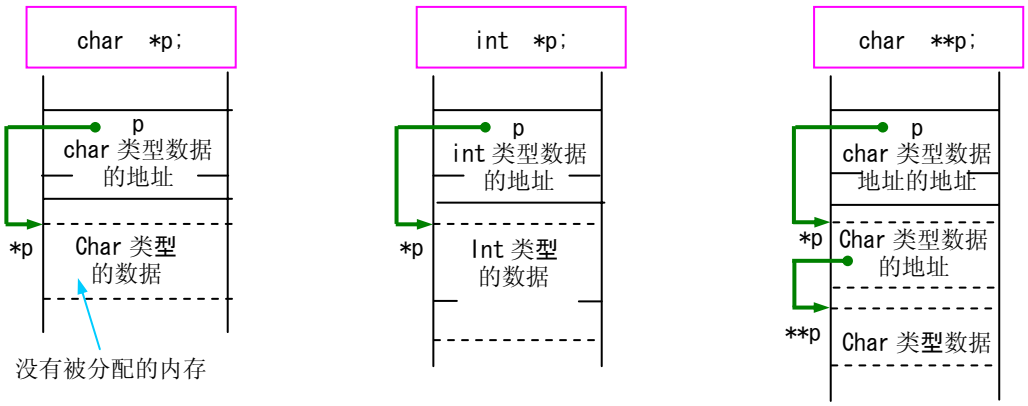
指针变量的声明

指针变量按下述格式进行声明。

```
数据类型  *指针变量名;
```

但是,上述声明只分配存放地址的内存。为确保数据体的内存,需要另外进行声明。

・指针变量的声明



指针与变量的关系

下面举例介绍指针变量与变量的关系，通过使用指针变量 p 将常数 5 赋值于 int 型变量 a。

```
void  main(void)
{
    int    a = 0 ;
    int    b ;
    int    *p ;

    p = &a ;
    *p = 5 ;
    b = a ;
}
```

地址运算符

「&a」表示变量 a 的地址
「*p」表示变量 a 的内容

A 变成了'5'

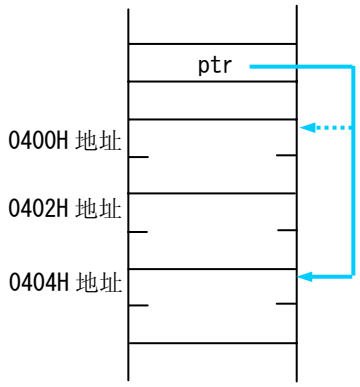
指针变量的运算

指针变量可以进行加法运算和减法运算。但是，指针变量运算与整数运算不同，运算结果是地址值。因此，指针变量因指示数据的长度不同，计算的地址值也不同。

```
地址  +  (整数  x  sizeof(型) )
地址  -  (整数  x  sizeof(型) )
```

```
int  * ptr;

ptr = (int * )0x0400;
ptr = ptr + 2;
```

指针变量 ptr 是指 int 型的变量。
int 型变量的长度用 sizeof(int)求字节长度，
结果为 2 字节。
因此，为 0404H 地址值为 ptr + 2xsizeof(int)。



专栏　指针变量的数据长度？

C 语言程序中变量的数据长度，由数据类型决定。指针变量其内容为地址值。因此，所使用的微处理器为指针变量准备了对应整个地址空间的数据类型。

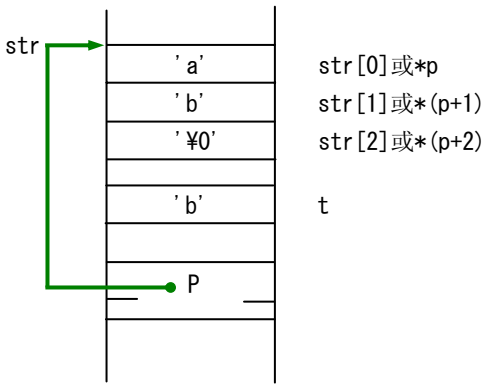
1.7.4 指针的应用

下面介绍指针的应用例。

指针变量与一维数组

数组名称加数字表示元素号码时,以变址寻址方式编码。因此,存取数组中的数据时,需要进行地址计算。
另一方面,利用指针变量时,成为间接寻址。

```
void main(void)
{
    char str[] = "ab";
    char *p;
    char t;

    p = str;
    t = *(p + 1);
          ¦
```

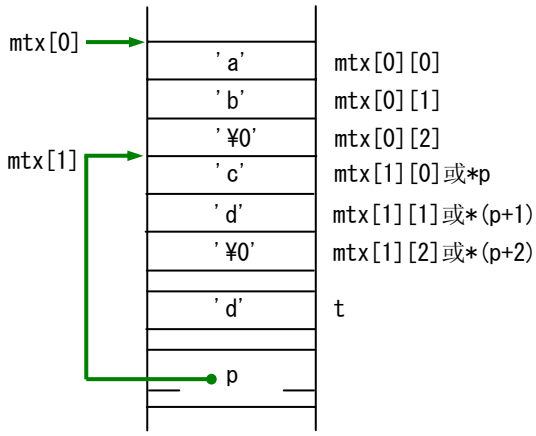


一维数组的开头地址可以从 str 取出
(地址运算符'&'省去)

指针变量与二维数组

二维数组与一维数组相同,可以使用指针变量存取数据。

```
void main(void)
{
    char mtx[2][3] = {
        "ab", "cd"
    };
    char *p;
    char t;

    p = mtx[1];
    t = *(p + 1);
          ¦
```



二维数组 mtx 第一行的开头地址可以从
mtx[1]取出('&'省去)

函数的地址传递

C 语言的函数间的基本的数据交换称值传递。但是使用这种方法,数组和字符串不能作为参数及返回值在函数间传递。

因此导入了利用指针变量的地址传递方法。该方法不但可用于数组和字符串的地址传递,将几个数据变为返回值时也可以使用。

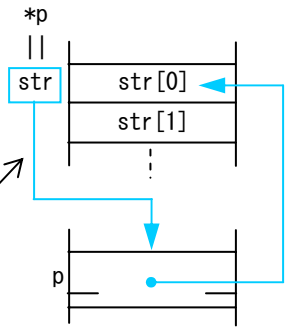
与单纯的值传递不同,地址传递中,被调用函数的指针变量的内容一旦被重写, 调用函数的数据领域可以直接被重写,因此降低了函数的独立性。

下面介绍通过地址传递来交换数组的方法。

<调用一方>

```
#define  MAX  5

void  cls_str(char *);

void main(void)
{
    char  str[MAX];
        :
    cls_str(str);
        :
}
```

数组的开头地址以参数形式传递



<被调用一方>

作为指针变量引用

```
void  cls_str(char *p)
{
    int  i;
        :
    for(i = 0; i < MAX; i++){
        *(p + i) = 0;
    }
}
```

操作数组的内容

专栏　函数间数据的快速传递

函数之间数据传递方法,除数值传递与地址传递以外,还有一种将传递数据声明为外部变量的方法。

因使用该方法时函数会失去独立性,所以不推荐在 C 语言程序中使用。调用函数时的入口处理和出口处理(参数与返回值的传递)可省略,具有可快速调用函数的优点。函数间数据的快速传递对函数的通用性没有要求,可用于 ROM 化程序。

1.7.5 指向数组的指针变量

下面详细介绍指针数组。

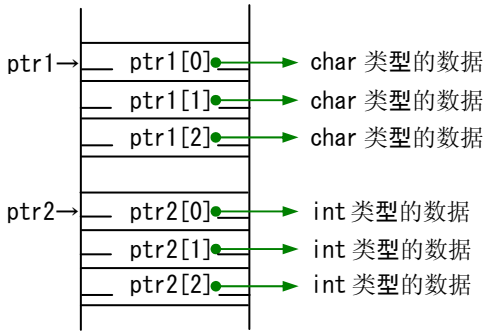
指针数组的声明

指针数组的声明方法如下所示。

数据类型　*数组名称［元素个数］;

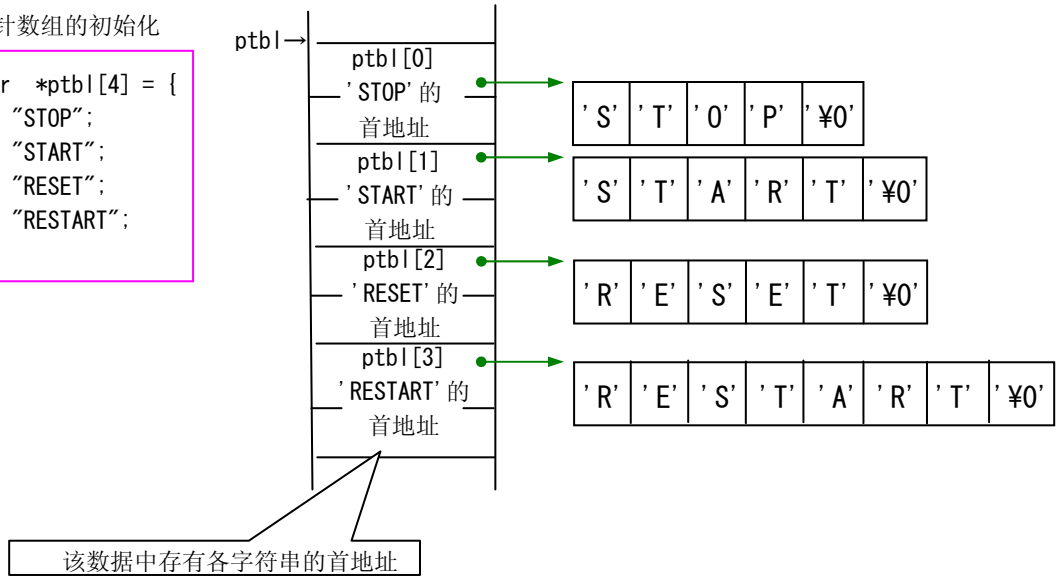
・指针数组的声明

```
char  *ptr1[3];
int   *ptr2[3];
```



・指针数组的初始化

```
char  *ptbl[4] = {
     "STOP";
     "START";
     "RESET";
     "RESTART";
};
```

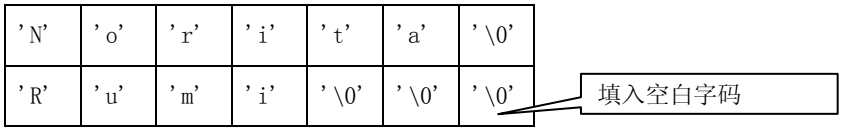


指针数组与二维数组

下面就指针数组与二维数组的不同点进行说明. 当声明长度不同的几个字符串形成的二维数组时, 空白区域填入空字码'\0'。同样情况下用指针数组进行声明时, 内存中则没有空白。

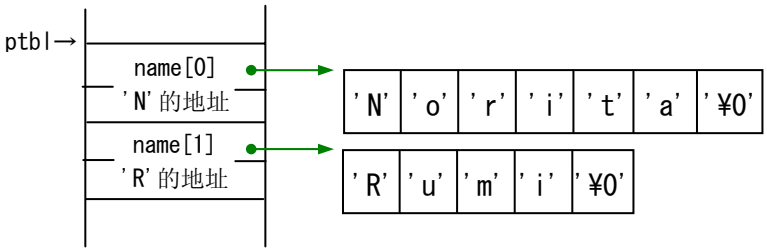
・二维数组

```
char name[2][7] = {
    "Norita",
    "Rumi",
};
```



・指针数组

```
char  *name[2] = {
    "Norita",
    "Rumi",
};
```



1.7.6 函数指针的使用例

汇编语言程序中,在数据需要切换处理比较多的情况下使用跳转表。同样情况下,借助上述的指针数组,用 C 语言也可以描述。

下面介绍使用函数指针来描述跳转表的方法。

函数指针是什么?

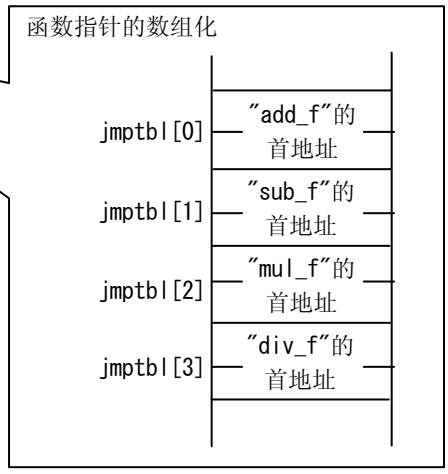
与前面所介绍的指针相同,指向函数的首地址的指针为函数指针。使用函数指针时,调用函数可以参数形式存在。声明与引用的格式如下。

<声明的格式> 返回值类型(*函数指针名称)(参数的数据类型);
<引用的格式> 保存返回值的变量=(*函数指针名称)(参数);

使用 Table Jump 进行四则运算的切换

变量"num"的内容选择操作方法。

```
/*  原型声明 **********/
int  calc_f(int, int, int);
int  add_f(int, int), sub_f(int, int);
int  mul_f(int, int), div_f(in, int);

/*  Jump Table  **********/
int  (*const jmptbl[4])(int, int) = {
    add_f, sub_f, mul_f, div_f
};

void  main(void)
{
    int  x = 10, y = 2;
    int  num, val;

    num = 2;
    if(num < 4){
        val = calc_f(num, x, y);
    }
}

int  calc_f(int m, int x, int y)
{
    int  z;
    int  (*p)(int, int);

    p =jmptbl[m];
    z =(*p)(x, y);
    return z;
}
```



函数地址设定

函数指针的引用

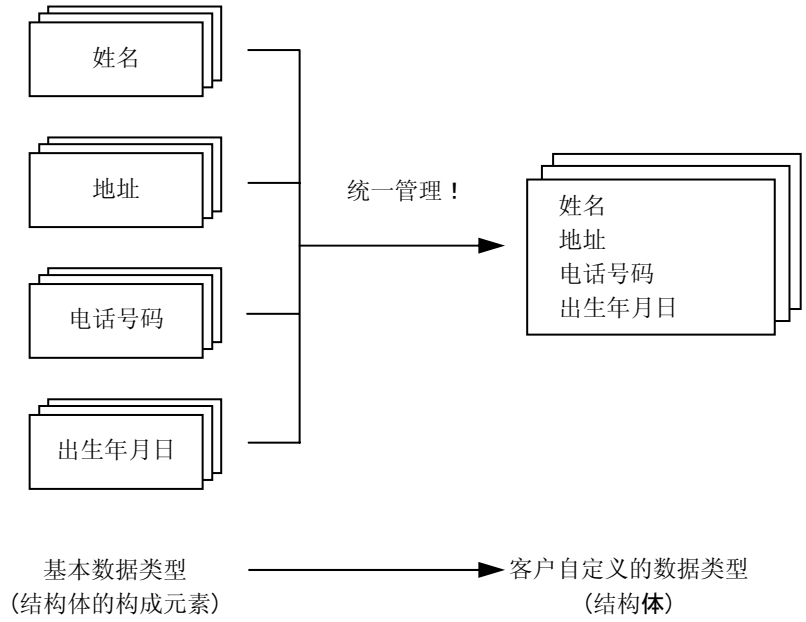
1.8 结构体与共用体

1.8.1 结构体与共用体

到目前为止介绍的数据类型(char 型,signed int 型,unsigned int 等),由编译规则决定,称为基本数据类型。
在 C 语言中,可以利用这些基本数据类型进行组合而构成新数据类型,称之为结构体与共用体。
下面介绍结构体与共用体的声明方法及引用方法。

从基本数据类型到结构体

使用结构体与共用体,在基本数据类型的基础上,按照要求组成客户自定义的数据类型。而且,新组成的数据类型可以和基本数据类型一样引用或构成数组。



1.8.2 新数据类型的构成

构成新数据类型的基本元素称为成员。要构成新的数据类型,需要对成员组成进行声明以及内存分配,必要时可以引用。下面分别介绍结构体与共用体的定义及引用方法。

结构体与共用体的区别

结构体与共用体在确保内存区域时,成员的配置方法不同。
(1)结构体:成员按顺序配置。
(2)共用体:成员配置在同一地址中。
(几个成员共用一个内存区域。共用体的长度采用分配在同一地址中成员的最大长度)

结构体的定义

为定义结构体型,使用关键字"struct"。

```
struct  结构体名{
        成员 1 ;
        成员 2 ;
            :
};
```

结构体的数据类型按上述形式描述。如果使用这种数据类型对变量进行定义,与通常的变量相同, 可确保内存区域。

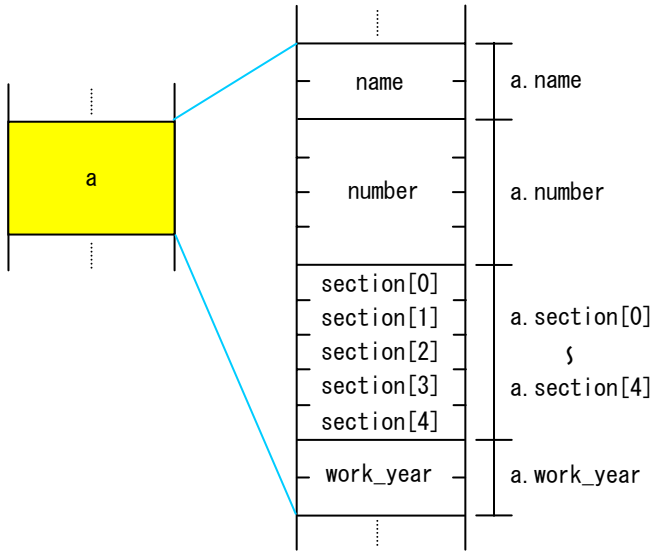
```
struct  结构体名   结构体变量名 ;
```

结构体变量的引用

引用结构体变量的各成员时,使用结构成员运算符'.'。

结构体变量名.成员名

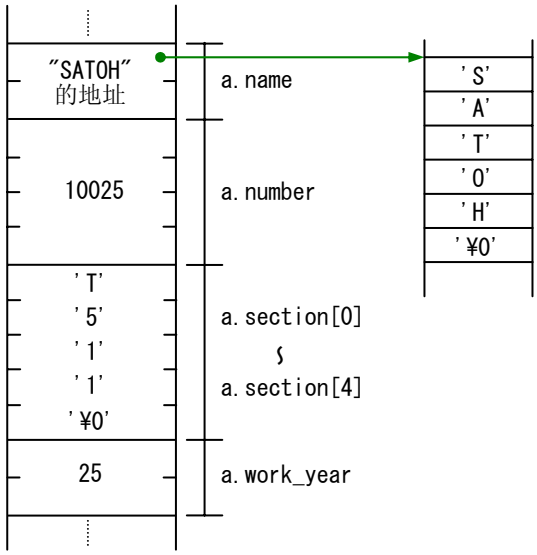
```
struct  person{
    char  *name;
    long  number;
    char  section[5];
    int  work_year;
};

void  main(void)
{
    struct person a;
    a.name = "SATOH";
    a.number = 10025;
    a.section = "T511";
    a.work_year = 25;
        :
```



结构体变量进行初始化时,各成员的初始数据按声明的先后顺序排列。

・结构体变量的初始化

```
struct  person a = {
    "SATOH", 10025, "T511", 25
};
```

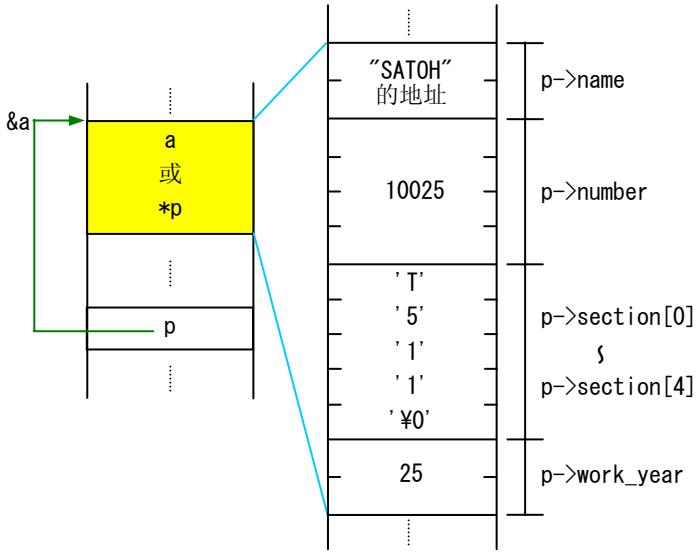


使用指针的引用例

使用指针引用各成员时使用「'->'」。

指针->成员名

```
#define  LYEAR  20
struct  person{
    char  *name;
    long  number;
    char  section[5];
    int  work_year;
};

struct  person  a = {
    "SATOH", 10025, "T511", 25
};

void  main(void)
{
    struct person *p;
    p = &a;
    if( p->work_year > LYEAR){
              ⋮
```

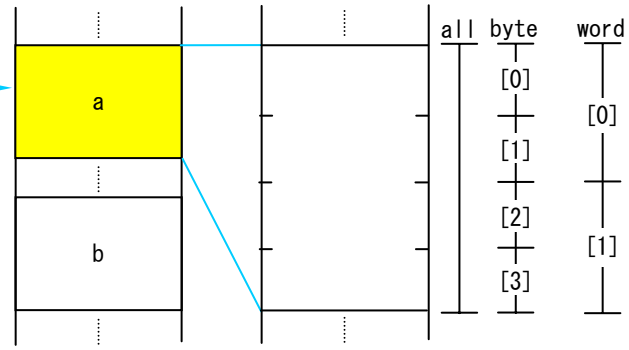


共用体

在共用体中,内存区域由所有成员共同使用。因此,可以将绝对不会同时存在的几个数据设定在一个共用体中,这样能节省内存使用量。而且根据不同情况可设 16 位长,8 位长等, 对于需要进行位切换的数据而言非常方便。

定义共用体时用"union"来描述。除此以外的定义,声明,引用均与结构体相同。

```
union  pack{
    long  all;
    char  byte[4];
    short  word[2];
};

void  main(void)
{
    union  pack  a, b;
        :
```



4 字节内存由 all, byte, word 共同使用

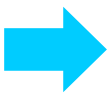
专栏　类型定义

结构体和共用体中需要使用关键字"struct"和"union",被定义数据类型的字符串就要增加。为了回避这种现象,使用类型定义 "typedef"。

```
typedef  现存类型名  新类型名;
```

如果按上述方式来记述, 新类型名与现存类型名等价,程序中哪个类型名都可以使用。下面介绍一下使用"typedef"的实例。

```
struct  data{
    char  a;
    short b;
    long  c;
};

struct data  sdata, *sptr;
        :
```



```
typedef struct{
    char  a;
    short b;
    long  c;
}DATA;

DATA  sdata, *sptr;
        :
```

进行类型定义时
可以省略结构体
(共用体)名

1.9 预处理命令

1.9.1 ICC740的预处理命令

C 语言中,以预处理命令(preprocess commend)支持读取文件,宏定义,条件编译等功能。
下面介绍 ICC740 中几个主要的预处理命令。

ICC740 的预处理命令一览表

为了与其他执行语句区分开,每个预处理命令的开头均带'#'。记述位置任意,而且分开不使用分号';'。
下表是在 ICC740 中可以使用的主要的预处理命令。

记述	功能
#include	包含文件。
#define	进行字符串置换及宏定义。
#undef	取消#define 的定义。
#if～#elif～#else～#endif	进行条件编译。
#ifdef～#elif～#else～#endif	进行条件编译。
#ifndef～#elif～#else～#endif	进行条件编译。
#error	将信息输出到标准输出,中断处理。
#line	指定文件的行号码。
#pragma	指示 ICC740 的扩展功能处理。

1.9.2 包含文件

读取包含文件时使用"#include"。检索目录不同,记述方法也有所不同。
下面介绍针对不同目的"#include"的描述方法。

检索标准目录

```
#include  <文件名>
```

读取编译操作 '-I' 所指定的目录中的文件。该目录中不存在文件的情况下,检索 ICC740 的环境常数"C_INCLUDE"所设定的标准目录,并读取文件。
通常,标准目录指含有「标准包含文件」的目录。

检索当前目录(Current Directory)

```
#include  "文件名"
```

读取当前目录中的文件。如果当前目录中不存在文件,按照编译操作'-I'中指定的目录,ICC740 的环境变量"C_INCLUDE"中设定的目录的检索顺序来读取文件。
为了与标准 include 文件区别,将独自作成的 include 文件放入当前目录中,使用这种描述方法来指定。

"#include"使用例

检索范围内所有目录中都没有该文件的情况下,输出 include error。

```
/*  include  *********/

#include  <stdio.h>

#include  "usr_global.h"

/*  主函数  **********/
void main(void)
{
      ¦
      ¦
}
```

从标准目录
读取标准包含文件

从当前目录
读取全局变量的头文件

1.9.3 宏定义

字符串的置换和宏定义中使用#define 识别符。为区分变量和函数,识别符一般使用大写字母。
下面介绍宏定义及其取消方法。

常量的定义

C 语言中常量前面可以加名称。这是减少程序中意义不明确的立即数,使定义通用的有效方法之一。

```
#define  THRESHOLD  100
#define UPPER_LIMIT  (THRESHOLD+50)
#define LOWER_LIMIT  (THRESHOLD-50)
```

阀值设为 100

上限设为+50

下限设为-50

字符串的定义

```
#define  TITLE  "Position control program"
char  mess[] = TITLE;
```

TITLE 的位置放入所定义的字符串

字符串加名称的情况下使用。

宏函数的定义

"#define"也可以用于对宏函数进行定义。宏函数与通常的函数相同,允许进行参数・返回值的传递。而且因为没有通常函数所需要的入口处理和出口处理,执行速度更快。
另外,对宏函数没有必要进行参数的数据类型宣言。

```
#define  ABS(a)  ((a) > 0 ? (a) : -(a))
```

返回参数绝对值的宏函数

```
#define  SEQN(a, b, c){¥
    func1(a) ; \
    func2(b) ; \
    func3(c) ; \
}
```

'\'表示连续描述
改行后描述的情况也按照连续字符串判断

几个语句时用'{'、'}'围起来

定义的取消

```
#undef  识别符
```

在"#define"中定义的识别符的置换,在"#undef"以后不能执行。
但是以下 8 个识别符为编译程序的保留字, 不要使用"#undef"。

- __FILE__ ; 源文件名称
- __LINE__ ; 现在的源文件的行号码
- __DATE__ ; 编译日期
- __TIME__ ; 编译时间
- __IAR_SYSTEMS_ICC__; ICC 编译程序
- __STDC__ ; ICC 编译识别符
- __TID__ ; 目标识别符
- __VER__ ; 编译程序的版本号码

1.9.4 条件编译

ICC740 允许在三种条件下对编译进行控制。根据规范进行函数切换时,或者控制是否嵌入调试用函数等情况下使用。
下面介绍条件编译的种类与介绍方法。

各种条件汇编

下面是 ICC740 中能使用的条件汇编的种类。

记述方法	内容
#if 常数式 A #else B #endif	常数式为真(不是 0)的情况下对 A 块进行编译, 不是真的情况下对 B 进行编译
#ifdef 宏名称 A #else B #endif	宏名称被定义的情况下对 A 块进行编译, 没有定义的情况下对 B 块进行编译
#ifndef 宏名称 A #else B #endif	宏名称没有被定义的情况下对 A 块进行编译, 被定义的情况下对 B 块进行编译

3 种类型中"#else"块均可省略。分三块以上的情况下,使用"#elif"追加条件。

识别符的定义

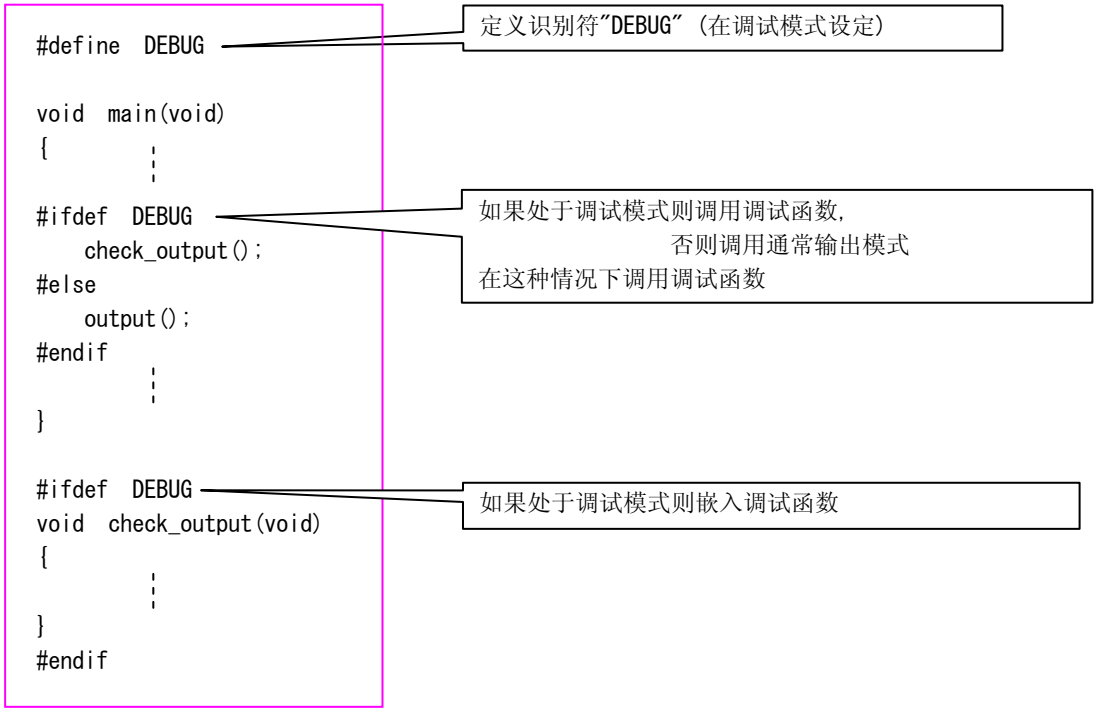
根据"#define"或 ICC740 的编译操作'-D'对识别符进行定义。

#define 识别符 ← 根据"#define"进行定义

%ICC740 -D 识别符 ← 根据编译操作进行定义

条件汇编例

下面是利用条件汇编控制调试函数的嵌入。



第 2 章
ROM化技术

本章以ICC740的扩展功能为中心介绍编写嵌入式程序时的注意点。

2.1 内存分配

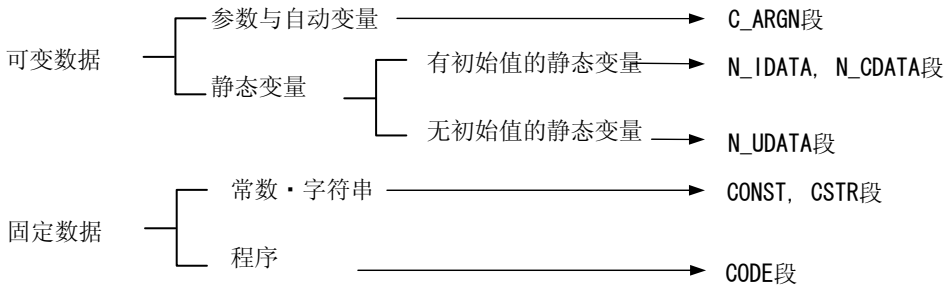
2.1.1 代码/数据的种类

组成程序的数据/代码有各种各样,有的能改写,有的不能,有的设定初始值而有的则不设定。必须根据所有数据/代码的不同性质将它们分配到ROM区,RAM区以及堆栈区。

下面介绍ICC740生成数据/代码的种类。

ICC740所生成的数据/代码

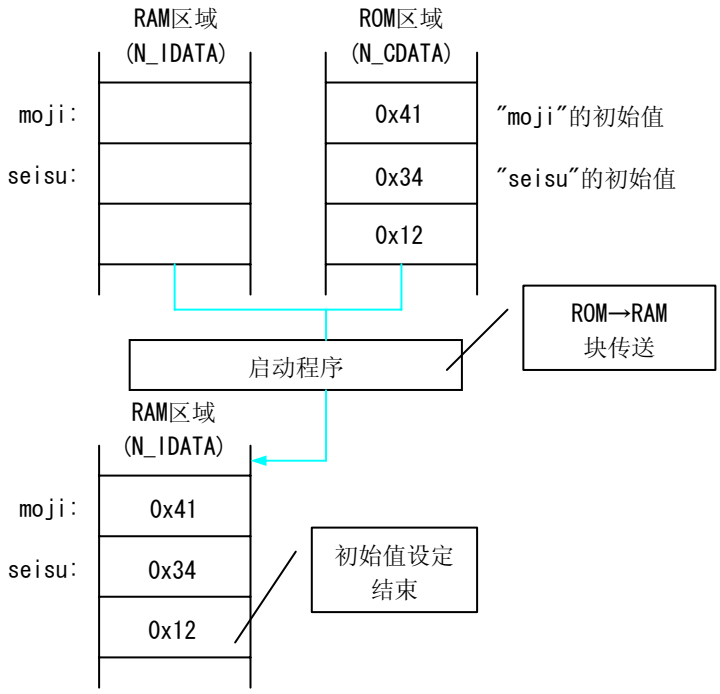
下面是ICC740所生成的数据/代码的种类与分配区域。



具有初始值的静态变量的使用方法

具有初始值的静态变量是能够可改写的数据,因而一定要分配在RAM上。
但是,这个变量没必要进行初始值设定。
在ICC740中,为带有初始值的静态变量在RAM上保有一定区域,而将它们的初始值保存在ROM上。
然后在启动程序时,将ROM上的初始值复制到RAM的保有区域上。

```
char  moji = 'A' ;
int  seisu = 0x1234 ;
void main ( void )
{
        . . .
}
```



2.1.2 ICC740所管理的区段

ICC740中数据/代码的配置区域以段来进行管理。
下面介绍ICC740生成且管理段的种类与管理方法。

段的构成

ICC740中将数据按种类划分在不同段中分别管理。下面是ICC740管理段的构成。

段名	内容	配置	
BITVARS	固定位变量存储区	RAM	Z
ZPAGE	库变量(分配在zero page)	RAM	Z
C_ARGZ	存储Auto变量,函数参数 (使用关键词zpage的变量)	RAM	Z
Z_UDATA	不带有初始值的外部变量(使用关键词zpage的变量)	RAM	Z
Z_IDATA	带有初始值的外部变量(使用关键词zpage的变量)	RAM	Z
EXPR_STACK	堆栈表达式 (尺寸在lnk740.xcl指定,在寄存器X操作)	RAM	Z
INT_EXPR_STACK	中断式堆栈(尺寸在lnk740.xcl指定,在寄存器X操作)	RAM	Z
CSTACK	通常的堆栈(尺寸在 lnk740.xcl 指定, 配置在 cstartup 指定)	RAM	Z/N
NPAGE	库变量(分配在 zero page 以外)	RAM	N
C_ARGN	存储 Auto 变量,函数参数	RAM	N
N_UDATA	存储没有初始值的外部变量	RAM	N
N_IDATA	存储有初始值的外部变量	RAM	N
ECSTR	可以写入的字符串 指定 ICC740 操作-y 时确保	RAM	N
RF_STACK	调用递归函数时的堆栈(尺寸通常为 0,使用时确保 256 字节以上)	RAM	N
RCODE	存储库函数代码(library code)	ROM	-
Z_CDATA	存储初始化常数(Z_IDATA 段用的初始值)	ROM	-
N_CDATA	存储初始化常数(N_IDATA 段用的初始值)	ROM	-
C_ICALL	调用间接函数表	ROM	-
C_RECFN	递归函数表	ROM	-
CSTR	存储常数字符串	ROM	-
CCSTR	存储指定 ICC740 操作-y 的初始值	ROM	-
CODE	存储程序代码	ROM	-
CONST	存储常数	ROM	-
C_FNT	特殊页・跳转表	ROM	-
INTVEC	中断向量表	ROM	-

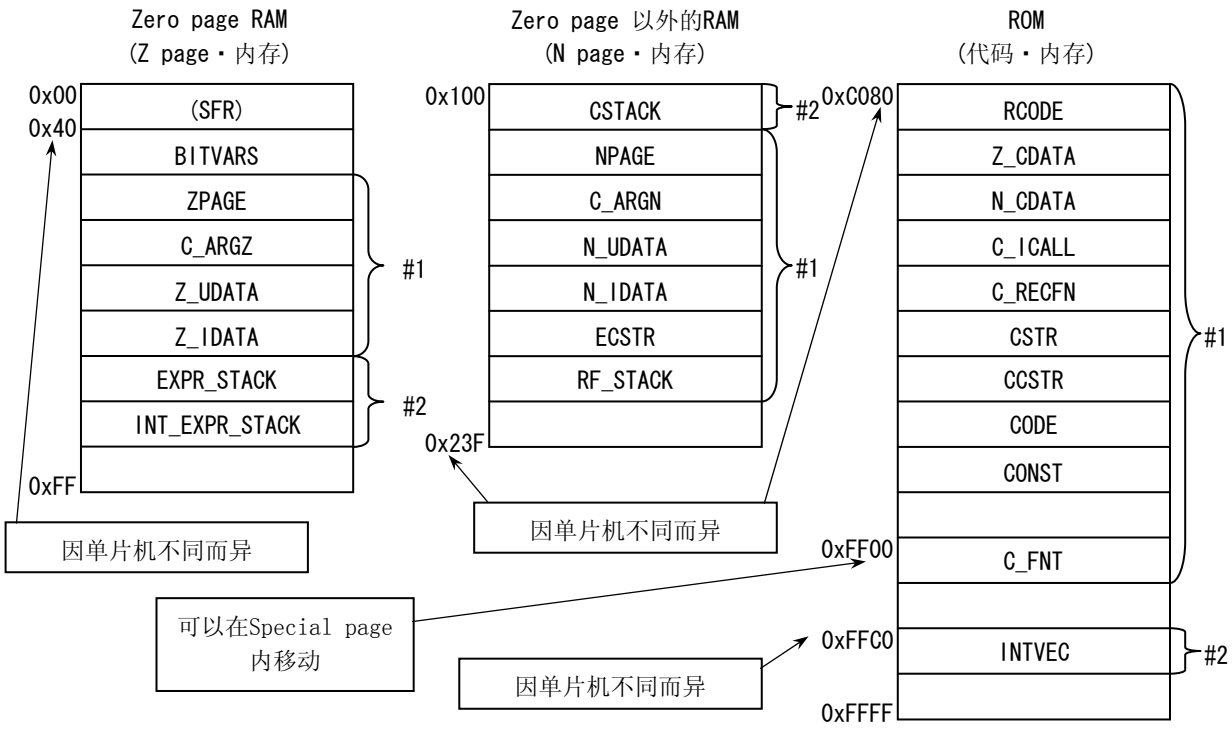
※Z ： 分配在 zero page 中
※N ： 分配在 zero page 以外
※Z/N ： 分配在 zero page 或 page 1

2.1.3 控制内存分配

ICC740可以结合用户的系统有效分配内存。

段的内存分配

ICC740 根据数据的种类对段的内存分配方法如下所示。



#1 的尺寸由 linker XLINK 设定。#2 的尺寸在 lnk740.xcl 文件中设定。
在CSTACK中，在page开头设256字节以下。这些依存于程序。

Large模式与Tiny模式

Large模式与Tiny模式中,变量的default设置位置不同。Large模式被设置在0 page以外，而Tiny模式的设置在0 page。

段名的追加("lnk740.xcl")

ICC740所生成的段由link・commend・file"lnk740.xcl"指定分配顺序。
新生成的段,必须在link・commend・file"lnk740.xcl"追加段名。

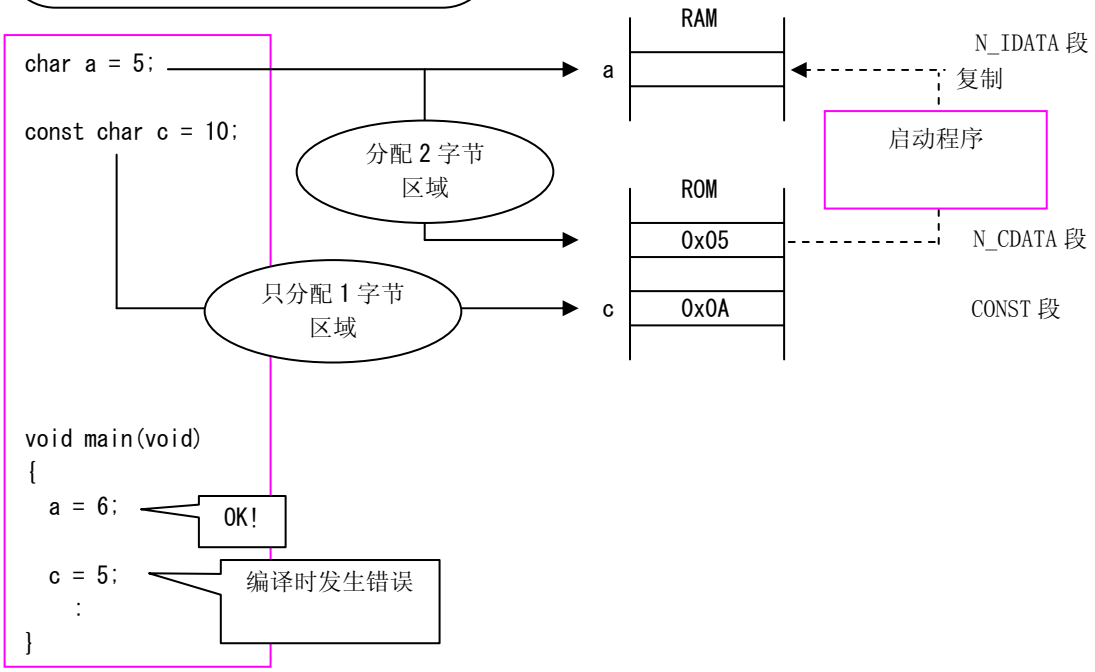
```
-!                  - lnk740.xcl -

        XLINK 4.44, 或更高, 将用于 740 的命令文件
        C-compiler V1.xx
        Usage:  xlink  your_file(s)  -f lnk740
                                          .
                                          .
                                          .
-!  Setup all read-only segments (PROM) at address 8000  -!
-Z(CODE)RCODE,Z_CDATA,N_CDATA,C_ICALL,C_RECFN,CSTR,CCSTR,CODE,CONST,NEW_CODE=C080-FEFF
                                          .
                                          .
                                          .
```

追加新生成的段名

强制将段分配在ROM区域(const修饰符)

在类型声明时,如果一个变量有初始值,则在RAM区域和ROM区域中分配内存。但是,该变量在程序运行过程中作为固定值不产生变化时, 类型声明时用const修饰符描述。这样,只在ROM区域分配内存而不使用RAM区域的内存,因而可以节省内存使用。另外,由于编译时直接赋值,因此可以检查出程序中是否存在改写ROM的语句。

```
const  数据类型  变量名=值  ;
```



专栏　改变段名

`#pragma　codeseg　(改变段名)`

改变ICC740所生成程序用的段名称。

```
void func1( void )
{
    . . .
}

#pragma codeseg ( NEW_CODE )

void func2( void )
{
    . . .
}
```

展开映像

```
    RSEG CODE
func1:
      . . .

    RSEG NEW_CODE
func2:
      . . .
```

func1 以 default 的段名展开,func2 以修改后的段名展开。
修改的段名,不能与编译器中保存的任一个段名重复。

除上以外,下例所示段名被修改的情况(指那些分配到有名称的段)。

`#pragma　memory=constseg　(段名)`　　常数分配在有名称的段

例

```
#pragma memory=constseg ( TABLE )
char ar[] = { 1, 3, -1, 5, -3 } ;      ← 将常数数组 arr 插入 ROM 段的 TABLE 中
#pragma memory=default                 ← 重新分配在 default 区域
```

注意,如果访问该常数数组时,该文件也必须进行同样的声明。

`#pragma　memory=dataseg　(段名)`　　将变量分配在有名称的段区内

例

```
#pragma memory=dataseg ( USRDAT )      ← 名为 USRDAT 的 RAM 区域内存放 3 个变量
char USRDAT_data1 ;
char USRDAT_data2 ;
int USRDAT_data3 ;
#pragma memory=default
```

如果,从别的文件访问该常数数组时,必须进行同样的 extern 声明。

`#pragma　memory=zpage`　　变量分配在 Z page 区域(0x00～0xFF)

例

```
#pragma memory=zpage
int buf[5] ;                           ← 变量 buf 和 f 存放到 ZPAGE 内存中
float f ;
no_init char *str[5] ;                 ← 将变量 str 强制地分配在 NO_INIT 内存中
#pragma memory=default
int a ;                                ← 变量 a 分配在 DATA 区域
```

2.2 初始化设定文件

2.2.1 初始化设定文件的作用

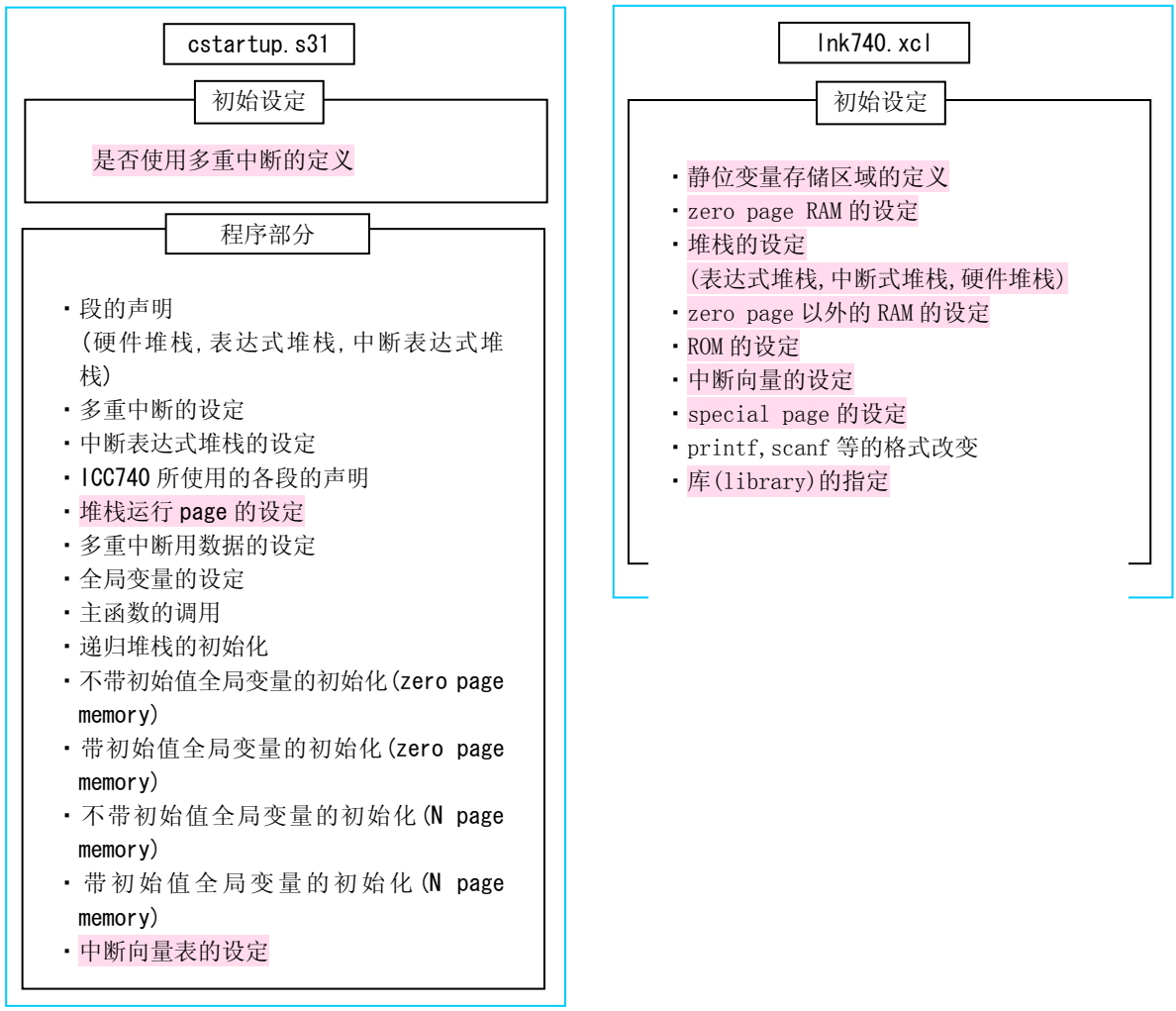
ICC740有两个初始化设定文件,启动程序"cstartup.s31"和link・commend・file"lnk740.xcl"。
下面介绍初始化设定文件的作用与构成。

初始化设定文件的作用

初始化设定文件的作用如下。
(1) 确保堆栈区域
(2) 单片机的初始设定
(3) 静变量区域的初始化
(4) 调用主函数
(5) 中断向量表的设定

初始化设定文件的构成

ICC740的初始化设定文件如下所示。



※ 涂色部分将在2.2.2节,2.2.3节详细说明。

2.2.2 启动程序

为了使嵌入式程序正常运行,操作前必须进行单片机的初始化和堆栈区域的设定。一般来说,这种操作不能用C语言来实现。与C言語源程序相区别,用汇编语言来编写初始设定程序,这就是启动程序。

下面介绍ICC740中的启动程序"cstartup.s31"。

启动程序的修改

请用户根据程序程序,进行下述修改。

cstartup.s31

- 定义是否使用多重中断
- 堆栈页的设定
- 中断向量的注册
- 复位向量(reset vector)的注册

是否使用多重中断的定义("cstartup.s31")

不使用多重中断的情况下,通过定义 NO_INTERRUPTABLE_ISR 段,RAM 1 字节,ROM 4 字节。
定义位置及描述方法如下所示。

```
;------------------------------------------------------------------
; Turning off 'interruptable ISRs':
; Do this if you need the extra byte(s)
;
; 1. Uncomment the define below
; 2. Assemble this file
; 3. Include the result in your linker command file:
;    -C cstartup.r31
;
; Variable '?IES_USAGE' and its initialization will no longer
; be included.
;------------------------------------------
;#define NO_INTERRUPTABLE_ISR
```

不使用多重中断的情况下,删除开头的";"。
使用多重中断的情况下,不要删除开头的";"。

堆栈页的设定("cstartup.s31")

进行堆栈页的设定时要注意 CPU 模式寄存器的各位的值一定要适合于使用环境。设定位置及描述方法如下所示。

```
;------------------------------------------------------------
;  RCODE - where the execution actually begins
;------------------------------------------------------------
    RSEG  RCODE:ROOT
init_C
        CLD                             ; set default mode
        CLT
        LDM  #0CH, 3BH                  ; set stack page : 3803 Group
        LDX  #LOW (SFE(CSTACK)-1)    ; set up stack pointer
        TXS
```

进行堆栈页的设定。设定为缺省(Default)中1page。
如果设定为 0 page,以 3803 组为例,需要进行下述设定。

LDM #08H, 3BH

中断向量与复位向量的注册("cstartup.s31")

附录 cstartup.s31 的 INTVEC 段的内容,根据使用单片机的中断领域进行变更,设**定**复位向量(复位后开始运行的模块名称)。

```
    COMMON  INTVEC

?CSTARTUP_INTVEC:
        BLKB    0FFFEH - 0FFDCH -2      ; 3803 Group
#if 0
#if defined(MELPS_37600)
        BLKB    40H - 6       ; FFFA  ( FFC0 + 40 - 6)  (-v2)
#elif defined(MELPS_MULDIV)
        BLKB    20H - 4       ; FFFC
#else
        BLKB    20H - 2       ; FFFE
#endif
#endif
?CSTARTUP_RESETVEC:
        WORD    init_C
        ENDMOD  init_C
```

#if～#endif 这一部分不使用。

复位后,从 init_C 开始运行。

在 BLKB 的领域设定中,请减去复位向量 (相当于上表中 -2 的部分)
INTVEC 段的首地址,记述在 lnk740.xcl 文件中。

2.2.3 链接命令文件

在链接命令文件中,进行与内存分配图（memory map）有关的详细设定。
下面详细介绍链接・命令・文件"lnk740.xcl"。

链接命令文件的修改

请结合程序,进行下述修改。

lnk740.xcl

- 段的配置与首地址/结束地址的设定
- 堆栈尺寸的设定
- 中断向量的首地址/结束地址的设定
- special page 的首地址/结束地址的设定
- 库的指定

段的分配与开头/结束地址的设定("lnk740.xcl")

ICC740 对生成段的分配及首地址/结束地址进行设定。对于没有指定首地址的段,将其分配在与以前所定义的段相连续的内存中。另外,用逗号区分开的段,按描述顺序分配在内存地址中。

下述段 (Segment) 将用于 ICC740,请不要删除。

```
-!  Setup "bit" segments (always zero if there is no need to reserve
    bit variable space for some other purpose) -!

-Z(BIT)BITVARS=200   -! address 40 (only) -!


-!  Setup "ZPAGE" segments.
    We allocate 41-FF for zero page by default.  It is assumed that
    00-3F is for SFRs while 40 is for a few "bit" variables.
    The following segment defintions (EXPR_STACK, INT_EXPR_STACK and
    CSTACK) that do not have an address given must fit inside the
    "41-FF" address range.
    If you have the CSTACK (processor stack) segme
    you have to give it an address and XLINK will no
    fit it within zero page.            -!

-Z(ZPAGE)ZPAGE,C_ARGZ,Z_UDATA,Z_IDATA=41-FF
                                        .
                                        .
-!  Setup "NPAGE" segments at address 1000-7FFF

-Z(NPAGE)NPAGE,C_ARGN,N_UDATA,N_IDATA,ECSTR=100-43F
                                        .
                                        .
-!  Setup all read-only segments (PROM) at address

-Z(CODE)RCODE,Z_CDATA,N_CDATA,C_ICALL,C_RECFN,CSTR,CCSTR,CODE,CONST=C080-FEFF
```

将 BITVARS 段在 0 page RAM 的开头进行定义。
位地址从 0 地址开始指定。
BITVARS=200 是指向 40h 的位地址。
(40hx8 位=200h 位地址)

zero page RAM 从 BITVARS 段后面的地址开始指定。
指定分配在 zero page 内存的段。

设定 zero page 以外的 RAM。
指定配置在 N age 内存的段。
如果 CSTACK 设定在 page 1,NPAGE 被分配在 CSTACK 从后面开始的地址中。

设定 ROM。
指定分配在 ROM 的段及其地址(除保留区域・中断向量区域外)。

※ -Z(XXX) ：在-Z 指定段的分配。XXX 用来指定内存的种类。在链接命令文件中,使用以下的类型。

- BIT 位内存
- ZPAGE zero page 数据内存
- NPAGE 通过绝对地址的指定而存取的数据内存
- CODE 代码内存

※ 段名 = YY(-ZZ) ：段分配在首地址 YY 开始的指定范围的段内。(ZZ 为结束地址)不需要指定范围的,只描述首地址。

堆栈尺寸的设定("lnk740.xcl")

使用堆栈的区域包括:进行复杂运算时使用的临时区域以及返回地址等。
ICC740 为充分使用内存,将堆栈划分为如下几种类型。

CSTACK	堆栈。 保留处理器堆栈。
EXPR_STACK	表达式堆栈。 在正常处理中对表达式进行评价时,临时保存结果。
INT_EXPR_STACK	中断式堆栈。 在中断处理中对表达式进行评价时,临时保存结果。
RF_STACK	递归(recursive)堆栈。 保存递归函数调用的局部变量与参数。

需要为堆栈设定一个合适的尺寸。如果堆栈尺寸太小,系统不能正常运行,严重一点会造成系统失控。而尺寸太大又会造成内存的无谓浪费。
下面介绍表达式堆栈,中断式堆栈,C堆栈的各尺寸的设定方法。上述堆栈所必要的尺寸在这个链接命令文件"lnk740.xcl"中设定。

```
-!  Setup "EXPR_STACK" segment. This zero page located stack is used
    to hold temporary when evaluating complex expressions.
    It is set to 20(hex) below. -!

-Z(ZPAGE)EXPR_STACK+20


-!  Setup "INT_EXPR_STACK" segment. This zero page located stack is used
    to hold temporary when evaluating complex expressions for interrupt
    routines written in C.  It is set to 20 below.
    You must give this stack space if you have C written interrupts that
    need an expression stack.  -!

-Z(ZPAGE)INT_EXPR_STACK+20


-!  Setup "CSTACK" segment. This is the CPU stack. Note that this can
    either reside in page 0 or 1  -!

-Z(NPAGE)CSTACK+40=100
```

设定表达式堆栈段。在缺省内其尺寸为 20h。

设定中断式堆栈段。在缺省中尺寸为 20h。

设定 C 堆栈段。在缺省内其尺寸在 100h-13Fh 范围内。

※ 段名+YY ：分配在段上的内存应与所设定的内存容量(YYh 字节)相同。
※ 将 C 堆栈 page 分配在 zero page 的情况下,请进行以下修改。
-Z(ZPAGE)CSTACK+40
在 INT_EXPR_STACK 后面分配 40h 字节。
*cstartup.s31 需要修改。

堆栈尺寸的确定方法

EXPR_STACK(表达式堆栈)

参考下面所使用的字节数，选择的堆栈尺寸要超过下述最大值。

- Short 类型，int 类型演算(乘法，除法)时 4 字节
- Long 类型运算(加法)时 8 字节
- Float 类型，double 类型运算(加法)时 16 字节
- 返回值的大小
  返回值不以变量代入而直接运算的情况下，加算该值。

INT_EXPR_STACK(中断式堆栈)

该堆栈值与 EXPR_STACK 取同值。

CSTACK(堆栈)

将下述各使用值相加，其和取为堆栈值

- 函数的最大嵌套数×2 字节(返回地址)
- 中断时的堆栈使用量
- 使用 MUL/DIV 命令时的堆栈使用量
- 汇编记述的堆栈操作量

中断向量的首地址/结束地址的设定("lnk740.xcl")

设定中断向量的首地址/结束地址。描述例如下所示。

```
-!  Setup the "INTVEC" interrupt segment.
    If you are using the 37600 (chip group -v2) and the default cstartup
    reset vector, you must change the INTVEC line below to:
        -Z(CODE)INTVEC=FFC0-FFFF
    If you have a tiny chip derivative that does not have the interrupt
    vectors in page FF, you can change the page of the addresses below.
    CSTARTUP inserts the reset vector relative to INTVEC start which
    means that you can change the page without any problems:
        -Z(CODE)INTVEC=1FE0-1FFF
        -Z(CODE)C_FNT=1F00

-Z(CODE)INTVEC=FFDC-FFFD
```

设定中断向量段。
指定中断向量的首地址到复位向量间的范围。

Special page 的首地址/结束地址的设定("lnk740.xcl")

设定 Special page 的首地址/结束地址。记述例如下。

```
-Z(CODE)C_FNT=FF00-FFDB
```

设定 Special page 段.
保留使用语言扩展功能 tiny_func 函数的地址。

库(Labrary)的指定("lnk740.xcl")

指定库(library)。

```
-! This example files selects the default library which is
   tiny memory model and a 740 with MUL/DIV.
   This corresponds to option -mt and -v0 to the compiler.
   If you want to use another library, you can do it by
   removing the comments around it and adding comments around
   the default library.

-C cl7400l

-! -C cl7400t -!              -!   -v0  -mt  -!
-! -C cl7400l -!              -!   -v0  -ml  -!
-! -C cl7401t -!              -!   -v1  -mt  -!
-! -C cl7401l -!              -!   -v1  -ml  -!
-! -C cl7402t -!              -!   -v2  -mt  -!
-! -C cl7402l -!              -!   -v2  -ml  -!
```

指定库。
选择**必要**的库,附带-C 操作**指定**

※ 库中包括有 default 的 cstartup 模块。如果不带-C 操作,将使用该 default 模块,请注意。
※ 库的种类

- cl7400t : 小模式
- cl7400l : 大模式
- cl7401t : 不带 MUL/DIV 的单片机, 小模式
- cl7401l : 不带 MUL/DIV 的单片机, 大模式
- cl7402t : 能访问带有扩张内存的单片机,小模式
- cl7402l : 能访问带有扩张内存的单片机,大模式

2.3 ROM化扩展功能

2.3.1 变量的分配

740族的最大存取空间为64K字节。ICC740将该区域划分Z page区域，其地址为0000h～00FFh，和N page区域,其地址为0100h～FFFFh地址。

下面介绍在该领域中变量,函数的分配方法及存取方法。

Z page区域与N page区域

在ICC740中,将最大存取空间64K字节划分为Z page区域和N page区域两个区域分别进行管理。
两个区域的特征如下。

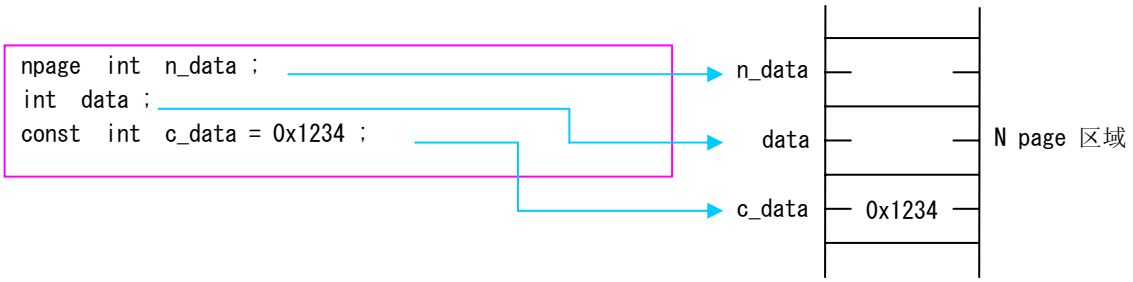
区域名	内容
Z page 区域	740族能够高效率快速存取数据的空间。堆栈和内部RAM分配在绝对地址0000h～00FFh的256字节区域中。
N page 区域	740族能够存取的绝对地址0100h～FFFFh的65280字节区域中,分配有堆栈,内部RAM和内部ROM。

变量的分配

类型指定符　变量名；

一般情况下,RAM数据分配在缺省的N page区域中。进行类型声明时指定在npage的数据,未指定在zpage的数据,用const修饰符修饰的ROM数据被分配在N page区域。

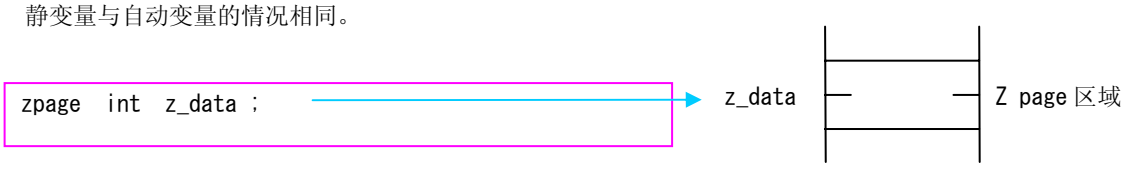
静变量与自动变量的情况相同。



Zpage中变量的分配

zpage　类型指定符　变量名；

类型声明时一旦指定zpage，RAM数据将被分配在Z page区域。
静变量与自动变量的情况相同。

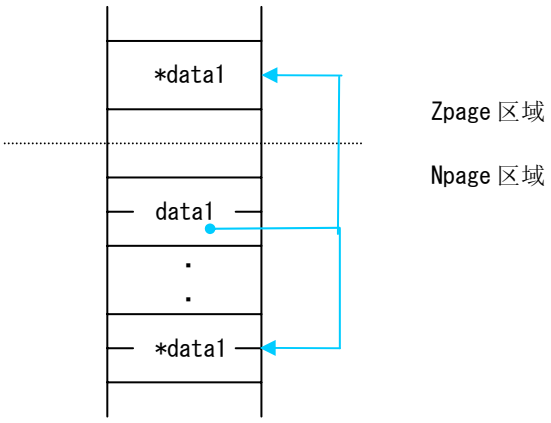


指针变量

指针变量自身和引用指针变量的地址都被分配在缺省区域(Npage区域)。另外,可以在缺省区域(Npage区域)中指定指针变量，而在Zpage区域指定调用地址。

类型指定符　＊变量名；

```
int *data1 ;
```



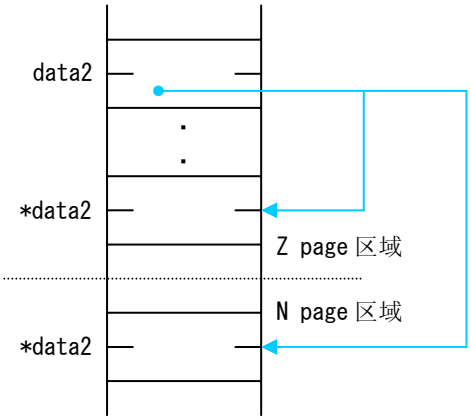
在 zpage 中指定指针变量

在zpage中指定指针变量,对存储在指针中地址长度以及指针自身的分配区域进行指定。

(1) 如果对指针自身分配区域进行指定情况下
指针变量自身被分配在 (Zpage 区域),引用地址被分配在 Zpage 区域或 Npage 区域,因数据而定。

类型指定符　＊zpage　变量名；

```
int *zpage data2 ;
```

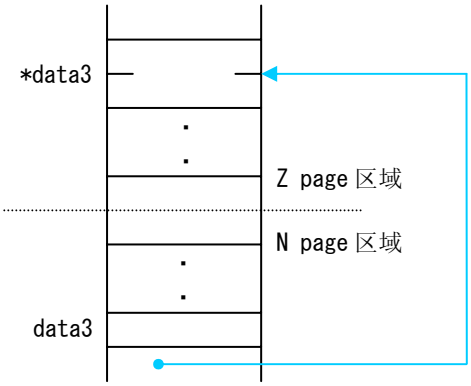


(2) 指定引用地址的分配区域的情况下
引用地址被分配在所指定的区域(Zpage 区域),指针变量自身被分配在缺省区域中(Npage 区域)。

类型指定符　zpage　＊变量名；

```
int zpage *data3 ;
```

然而,对这种描述会产生以下警告
Warning[w23]:Cannot represent pointer type
请只调用 Zpage 区域。



2.3.2 位处理

在ICC740中可以用位单位来处理数据,有两种方法来实现这种处理:一种是使用扩展功能,另一种是使用共用体。下面介绍具体方法。

位变量

ICC740中的位处理方法如下所示。

(1) 当使用扩展语言功能的位变量"bit"的情况下
位变量表示 Zpage sfr 变量(也就是 1 字节的 I/O 数据类型的变量)的 1 位。

```
sfr Port1 = 0x02 ;       // SFR 的地址 0x02 定义为"Port1"
bit Port14 = Port1.4 ;   // "Port1"的第 4 位定义为"Port14"
bit Port15 = 0x02.5 ;    // SFR 的 0002h 地址的第 5 位定义为"Port15"
```

(2) 使用共同体的情况下
共同体表示一个位类型结构体和字节变量。
Zpage,Npage 两者均可使用。

```
typedef union{
    unsigned char   byte ;  // 存取字节用
    struct{          // 位存取用
        char    _0 :    1 ;
        char    _1 :    1 ;
        char    _2 :    1 ;
        char    _3 :    1 ;
        char    _4 :    1 ;
        char    _5 :    1 ;
        char    _6 :    1 ;
        char    _7 :    1 ;
    }bitf ;
}bytestr ;

npage static bytestr    pre_t_5msec ;

void    main( void )
{
         pre_t_5msec.bitf._0 = 1 ;
}
```

生成分别对应 Zpage,Npage 的指令。

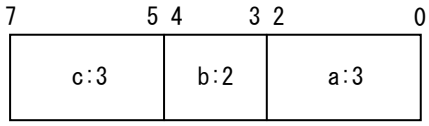
专栏　有关位字段(bit-fields)的配置

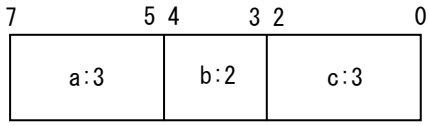
如果想改变位字段的存储顺序,则使用#pragma 伪指令。因为位字段的存储顺序取决于编译器,使用这种方法可以回避移植性问题(portability problem)。

#pragma bitfields=default　　恢复位字段存储顺序

#pragma bitfields=reversed　　反转位字段的内存顺序

```
struct{
        unsigned char  a : 3 ;
        unsigned chart b : 2 ;
        unsigned char  c : 3 ;
}bit1 ;

#pragma bitfields=reversed

struct{
    unsigned char   a : 3 ;
    unsigned char   b : 2 ;
    unsigned char   c : 3 ;
}bit2 ;
```





2.3.3 I/O界面的控制

嵌入式系统对I/O界面进行控制时,对变量指定绝对地址。ICC740指定绝对地址的方法有两种:一种是定义SFR区域,另一种是利用指针。

下面介绍各方法的详细内容。

定义特殊功能寄存器(SFR)区域

使用sfr变量的语言扩展功能来设定SFR区域。SFR设定文件通常作为一个独立文件放入源程序中。

下面是SFR定义文件的例子。

SFR定义文件<sfr_3803h.h>

```
    :
sfr PRE12   = 0x00020 ;
sfr T1  = 0x00021 ;
sfr T2  = 0x00022 ;
sfr TM  = 0x00023 ;
sfr PREX    = 0x00024 ;
sfr TX  = 0x00025 ;
sfr PREY    = 0x00026 ;
sfr TY  = 0x00027 ;
sfr TZL = 0x00028 ;
sfr TZH = 0x00029 ;
sfr TZM = 0x0002a ;
    :
```

设定绝对地址

<源文件>

```
#include    "sfr_3803h.h"
          :
void    main( void )
{
    :
    TX  = 0x94;
    TM.3 = 0 ;
    :
}
```

读取 SFR 定义的文件

参照 SFR 定义存取字节

参照 SFR 定义存取字节

通过指针指定绝对地址

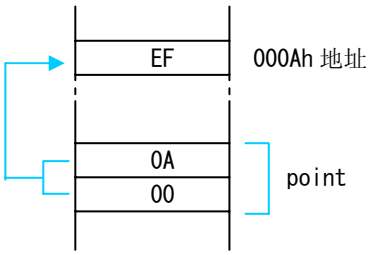
利用指针可以指定绝对地址。下面举例说明。

例)将0xEF 赋值于000Ah地址

```
char  * point ;
point = (char *)0x000A ;
*point = 0xEF ;
```

|| 如果整理为 1 行

```
* (char *)0x000A = 0xEF ;
```



2.3.4 无法使用C语言编程时的对策

对于「直接控制C标志」等有关硬件的问题,有时用C语言无法编写。在这种情况下, ICC740允许将汇编语言直接嵌入C语言的源程序中(这就是inline assemble feature方法)。Inline assemble方法包括使用inline函数与使用asm函数的两种方法。
下面详细说明这两种方法。

汇编指令的生成(inline 函数)

Inline函数中包括下述7种函数。

```
break_instruction() :   生成 BRK 指令,接下来是 NOP 指令
cld_instruction()   :   生成 CLD 指令
disable_interrupt() :   生成 SEI 指令
enable_interrupt()  :   生成 CLI 指令
enter_stop_mode()   :   生成 STP 指令
enter_wait_mode()   :   生成 WIT 指令
nop_instruction()   :   生成 NOP 指令
```

下面举例说明。

```
#include  <intr740.h>
void    main( void )
{
    :
    enable_interrupt() ;
    :
}
```

展开
映像

```
main:
    :
        CLI
    :
        RTS
```

※ 使用 inline 函数时,请安装 intr740.h。该头文件安装在 ICC740 的 inc 文件夹中。

只有一行汇编语言程序时(asm函数)

```
__asm("字符串") ;
```

采用上述形式描述时,用"(双引号)围起来的字符串(包括空白,Tab)被原封不动地嵌入到由C生成的汇编语言源程序中。因为这种描述在函数内部及外部均可进行,当需要直接操作标志,寄存器以及有必要进行快速处理时使用这种方法。
举例说明如下。

```
void    main( void )
{
    :
    __asm("  LDA #0") ;
    :
}
```

※描述__时并排使用两个_(underscores)。

2.4 与汇编语言链接

2.4.1 函数间的调用

汇编语言的子程序也可以从C语言程序中调用。从C语言程序中调用汇编语言程序的情况下,请按照C语言的调用步骤执行。下面详细介绍ICC740函数界面。

函数的入口处理与出口处理

在ICC740中调用函数,返回时主要进行下述两步处理。

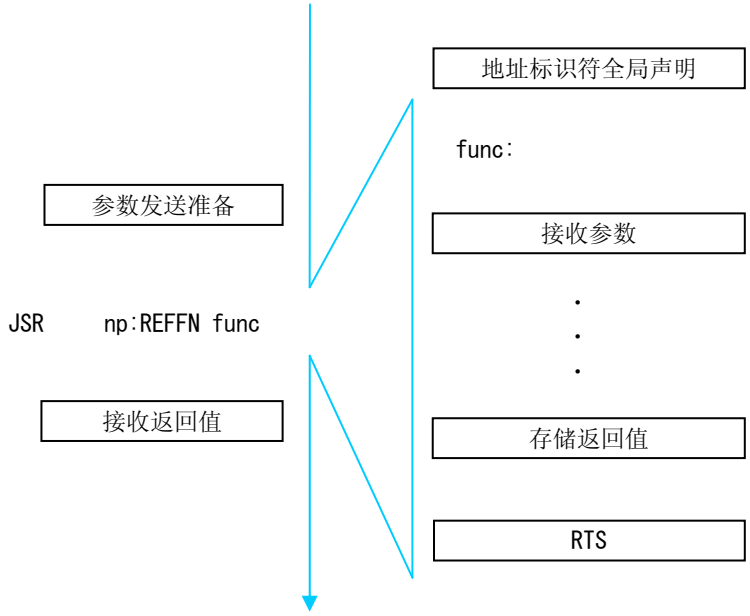
(1) 与函数进行参数的传递

(2) 与函数进行返回值的传递

上述操作的顺序如下。

```
int func( int, int ) ;

void main( void )
{
    int a = 3, b = 5 ;
    int c ;
        :
    c = func( a, b ) ;
        :
}
```

```
int func( int x, int y )
{
        :
}
```



参数的传递规则

ICC740中,根据参数的数据类型不同,参数的存储地点不同。下面是向函数传递参数的方法。

参数的类型	传递方法
Char 类型	累加器 ※
其他类型	C_ARGN 或 C_ARGZ

※ 只对第一个参数

返回值的传递规则

ICC740中,返回值的数据类型不同,其存储位置也不同。返回值的传递规则如下所示。

数据类型	返回方法
Char类型	累加器
其他类型	EXPR_STACK

堆栈的种类

ICC740中,调用函数时的参数及返回地址存储在下述段中。因此,ICC740不需要创建堆栈帧。

	段
参数	C_ARGN(zero page 以外) C_ARGZ(zero page)
返回地址 寄存器入栈	CSTACK

函数符号向汇编语言的变换规则

ICC740中符号的变换方法完全相同,与函数的性质无关。符号变换规则如下所示。

函数名	汇编符号名
例 func()	函数名: 例 func:

界面的关键字

对于从C语言中调用的汇编语言函数进行声明时,使用汇编界面关键词"DEFFN"。另外,从C语言程序调用汇编语言程序时,汇编语言程序的函数名需进行"PUBLIC"声明。之后, C语言程序内中,调用汇编语言程序的函数名需进行"EXTERN"声明。

```
DEFFN 函数名( a, 0, b, 0, 0x8000+x, 0, y, 0 )
```

a ：C_ARGZ段中设定函数的zero page自动变量的尺寸
b ：C_ARGN段中设定函数的zero page以外的自动变量的尺寸
x ：C_ARGZ段中设定函数的zero page参数的尺寸
y ：C_ARGN段中设定函数的zero page以外的参数的尺寸

程序的描述例如下所示。

```
extern void sub( void ) ;
void func( void )
{
    sub() ;
}
```

a.c

```
    DEFFN sub( 0, 0, 0, 0, 0x8000, 0, 0, 0 )
    PUBLIC sub
    RSEG P:CODE
sub:
    ・・・
    RTS
```

寄存器 X 与标志位的保存/恢复

b.s31

※ 因汇编语言的函数没有参数,a、b、x、y 必须为 0。
※ DEFFN 是 linker 计算段 C_ARGZ、C_ARGN 的尺寸时所必须的关键词。

2.4.2 从C语言中调用汇编语言程序

下面介绍用C语言函数调用汇编语言子程序的具体描述方法。

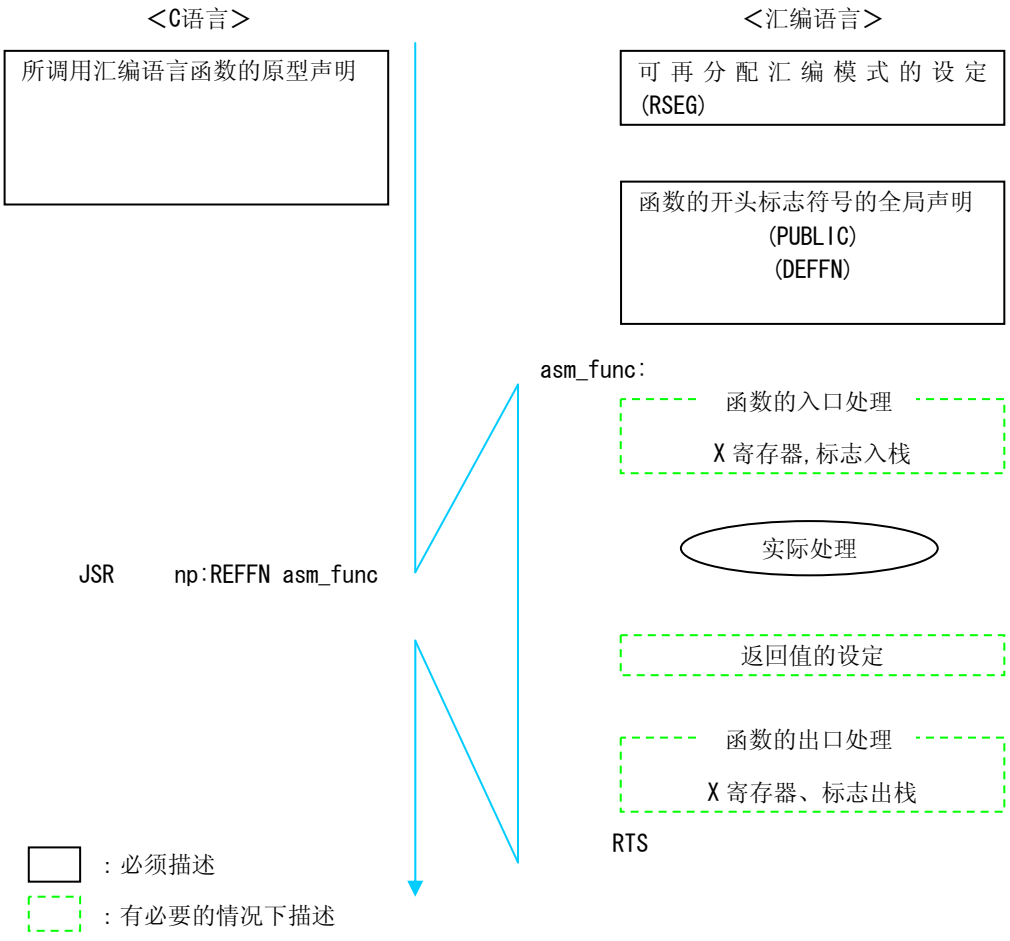
调用汇编语言子程序

从C语言程序中调用汇编语言子程序（汇编语言函数)时,请遵守以下的规则。

(1) 子程序与C语言程序用独立文件保存。
(2) 子程序名请遵循符号变换规则。
(3) 在调用程序一方的C语言程序中进行子程序(汇编语言函数)的原型声明。
此时,在存储类型说明符"extern"进行外部引用声明。
(4) 通常,子程序(汇编语言函数)中ICC740所使用的X寄存器,标志位不改变。
需要进行修改的情况下,在函数的入口将值压入栈，在函数出口将压入堆栈的值出栈。

```
extern void asm_func( void ) ;
void func( void )
{
    asm_func () ;
}
```

```
    DEFFN asm_func ( 0, 0, 0, 0, 0x8000, 0, 0, 0 )
    PUBLIC asm_func
    RSEG P:CODE
asm_func:
    ・・・
    RTS
```



调用子程序例

下面介绍用LED显示示计数结果的程序例。LED显示部分用汇编语言,计数部分用C语言作成并进行链接。

<计数部分>

```
void    led( void ) ;
sfr     P7 = 0x40 ;
extern char    counter ;

void    main( void )
{
    if( counter < 9 ){
        counter++ ;
    } else {
        counter = 0 ;
    }

    led() ;
}
```

<LED 显示部分>

```
PUBLIC    counter
PUBLIC    led
DEFFN     led( 0, 0, 0, 0, 32768, 0, 0, 0 )

RSEG      CODE
led:
    LDY    np:counter
    LDA    np:table, Y
    STA    zp:64
    RTS

RSEG      CONST
table:
    BYTE    0xC0
```

2.5 中断处理

ICC740允许用C语言函数来描述中断处理,具体按下面4个步骤来实现。

(1) 中断处理函数的描述
(2) 中断禁止标志 (I标志)的设定
通过inline函数进行。
(3) 中断向量表的注册
(4) 中断向量段的设定

2.5.1 中断处理函数的描述例

在这一节中,当3803群MCU每发生一次INT0中断(上升沿)和INT2中断(下降沿)时计数器 "counter"的内容就要清”0”。下面借用此例来介绍计数程序(count up program)。

描述中断处理函数

源文件的描述例。

```
#include    <intr740.h>          /* inline 函数用头文件 */
#include    "sfr_3803h.h"          /* 3803H 群用 SFR 头文件 */

unsigned char    counter ;

interrupt[30] void INT0_TimerZ( void )    /* 中断处理函数 */
{
    cld_instruction() ;          /* I0 进制模式标志的初始化 CLD 指令 */
    counter = 0 ;
}

Interrupt[8] void Int2( void )    /* 中断处理函数 */
{
    cld_instruction() ;          /* I0 进制模式标志的初始化 CLD 指令 */
    if( counter < 9 ){
        counter++ ;
    } else {
        counter = 0 ;
    }
}

void main( void )
{
    /* ①设定中断沿选择位以及中断源位 */
    INTEDGE.0 = 1 ;          /* INT0 上升沿有效 */
    INTEDGE.3 = 0 ;          /* INT2 下降沿有效 */
    INTSEL.0 = 0 ;           /* 中断源 → INT0 中断 */

    /* ②插入一个以上指令以后,将中断要求位清 0(不要求) */
    nop_instruction() ;      /* 插入一个指令 */
    IREQ1.0 = 0 ;            /* INT0 中断请求位 → 不要求*/
    IREQ2.3 = 0 ;            /* INT2 中断请求位 → 不要求 */

    /* ③将中断允许位设为“1”(允许) */
    ICON1.0 = 1 ;            /* INT0 中断允许位 → 允许 */
    ICON2.3 = 1 ;            /* INT2 中断允许位 → 允许*/

    enable_interrupt() ;     /* 中断允许位 CLI 指令 */

    while( 1 ) ;             /* 中断等待循环 */
}
```

如果完全不使用程序内 I0 进制模式标志,没必要在中断处理函数内进行初始化。

在"sfr_3803h.h"定义

```
sfr INTSEL    = 0x00039;
sfr INTEDGE   = 0x0003a;
sfr IREQ1     = 0x0003c;
sfr IREQ2     = 0x0003d;
sfr ICON1     = 0x0003e;
sfr ICON2     = 0x0003f;
```

ICON1.0 表示 SFR ICON1 的位 0。

处理器状态寄存器

(中断时自动入栈。)

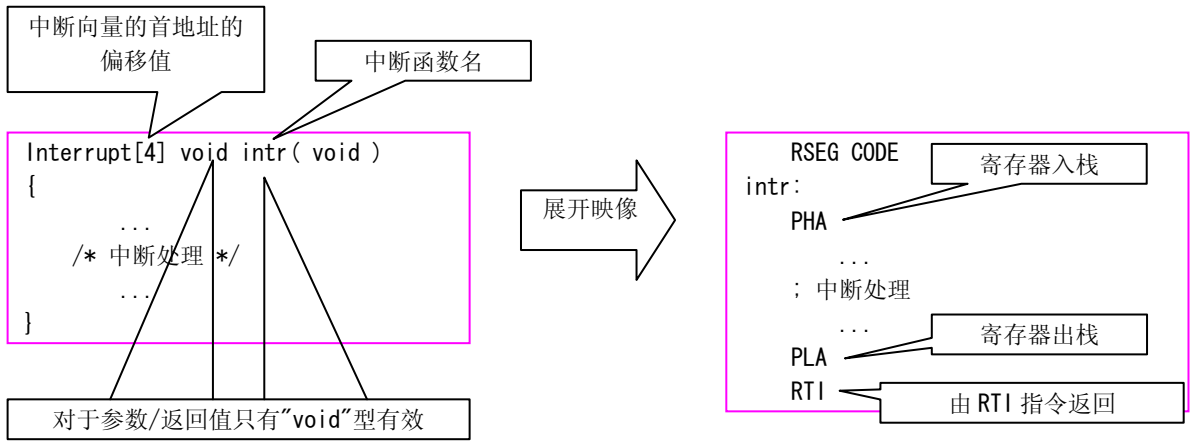
- D,T 标志
  在 cstartup.s31 的 init_C 进行初始化。
- I 标志
  复位后,成为 1(**禁止**)状态。

2.5.2 中断处理函数的描述方法

ICC740中,中断处理函数可以用C语言或汇编语言来描述。
下面介绍使用C语言来描述中断处理函数的基本方法。

中断处理函数描述的基本方法(C 语言)

中断处理函数的定义,使用扩展关键词 interrupt 来描述。描述方法与展开图如下所示。



按上述形式描述时,在函数的入口与出口,除正常的函数处理顺序之外,还有 740 族寄存器的入栈,出栈及生成 RTI 指令。

注意,入栈和出栈的寄存器的数量因中断处理函数的内容而不同。

另外,紧接着”interrupt”后面的括号内,指定中断向量的首地址的偏移值,中断处理程序的地址被插入到该向量中,发生中断时调用该函数。不指定偏移值时,中断函数的向量表在 cstartup.s31 文件中定义(用伪命令”EXTERN”对中断处理函数进行外部引用声明,然后注册在中断向量中。)

※ 对于参数 / 返回值,中断处理函数中只有 void 型函数有效。声明为其他类型时,编译时会发生错误。

使用中断时的注意事项

通常情况下调用的函数不要在中断函数内调用。
〈理由〉: 自动变量处于静状态,通常调用函数在被调用时一旦产生中断,该函数的自动变量将被改写。

中断函数不调用通常所调用的函数的情况下,同样的函数要准备两个:通常情况下用和中断用。

间接调用的函数,也存在与上述同样的限制。

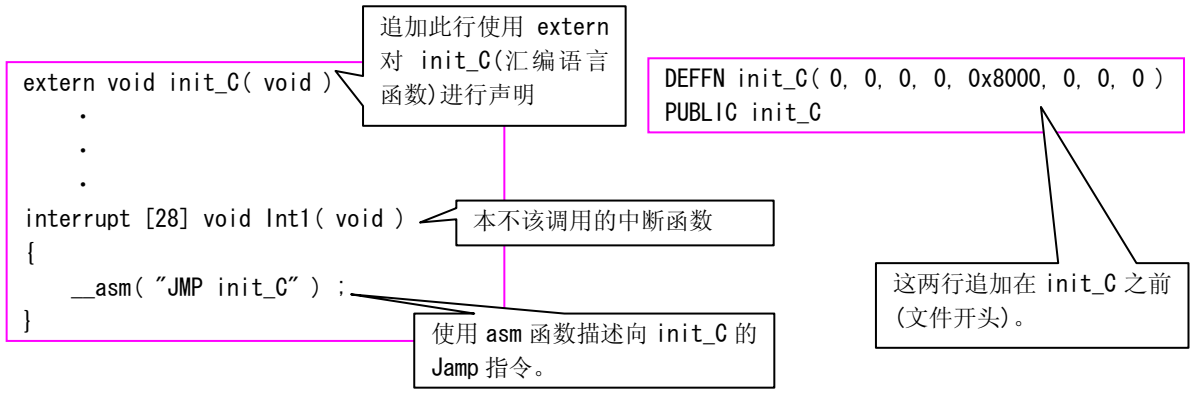
专栏 从中断处理函数向 init_C(启动)的跳转方法

对于不使用的中断函数，如下所示，对函数不作描述,括号内为空白。

```
interrupt [28] void Int1( void )
{
}
```

这种记述形式只生成 RTI 指令。因此,在突然产生中断的情况下，RTI 指令被执行后,程序开始继续执行。

除上述方法以外，在突然产生中断的情况下，与复位时的情况相同，从 init_C 开始执行程序。其记述方式如下:



2.5.3 中断禁止标志(I标志)的设定

中断禁止标志(I标志)的设定

要产生中断,中断禁止标志(I标志)一定要通过CLI指令设定为0(允许)。ICC740中,通过inline函数进行I标志设定。使用inline函数时,请确认"intr740.h"一定要包含在内。

```
#include    <intr740.h>

void main( void )
{
    enable_interrupt() ;
    while( 1 ) ;
    disable_interrupt() ;
}
```

展开映像

```
main:
    CLI          <-- enable_interrupt()
?0003:
    BRA    ?0003
    SEI          <-- disable_interrupt()
    RTS
```

在上例中,"enable_interrupt()"将被 I 标志置 0 的"CLI"指令、"disable_interrupt()"将被 I 标志置 1 的"SEI"指令所置换。

2.5.4 中断向量表的注册

中断处理函数的注册

中断处理函数的注册,是根据所使用的单片机,对 cstartup.s31 的 INTVEC 段的内容进行修改。

```
    COMMON  INTVEC

?CSTARTUP_INTVEC:
        BLKB    0FFFEH - 0FFDCH -2      ; 3803 Group
#if 0
#if defined(MELPS_37600)
        BLKB    40H - 6       ; FFFA  ( FFC0 + 40 - 6)  (-v2)
#elif defined(MELPS_MULDIV)
        BLKB    20H - 4       ; FFFC
#else
        BLKB    20H - 2       ; FFFE
#endif
#endif
?CSTARTUP_RESETVEC:
        WORD    init_C
        ENDMOD  init_C
```

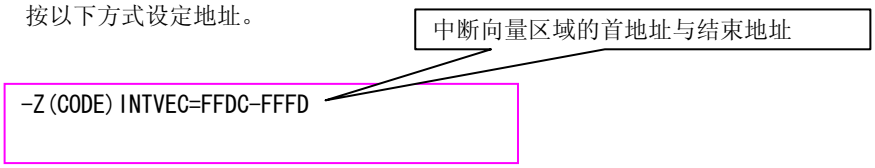
使用 BLKB 伪命令对中断向量区进行地址空间分配。因复位不是刻意造成的中断,指定地址空间中要减去复位向量部分的 2 字节。

复位向量设定在最下方。复位时,程序从复位向量中的 init_C 开始启动。

2.5.5 中断向量段的设定

中断向量段的设定

设定中断向量段时,结合单片机向链接命令文件"lnk740.xcl"中的中断向量段"INTVEC"中按以下方式设定地址。



※ 中断向量领域的首地址及结束地址,请设定与各单片机所对应的值。

2.5.6 多重中断的使用方法

下面介绍使用多重中断时所必要的设定,注意事项以及函数的描述方法。

使用多重中断时的注意事项

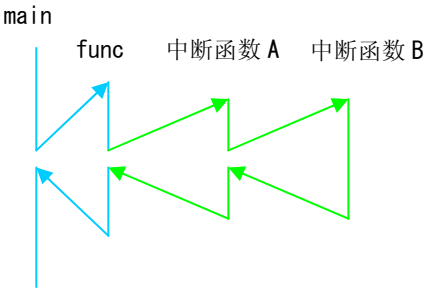
在下面所示的程序编程时，对中断函数 B 要注意下面几点:

- 标志(Flag)设定,count 更新等简单处理。
- 不要调用 C run time libratory。
- 返回值设定 char 类型。

＜理由＞

如果调用 C run time libratory 或调用返回值为 char 类型以外的函数时,则使用中断堆栈。

中断表达式堆栈是在中断处理中进行表达式评价过程中、临时保存结果用的区域。该区域所有的中断均可使用。因此,中断函数 A 所使用的中断表达式堆栈中的内容将被中断函数 B 的中断表达式堆栈内容覆盖(overwritten)。



编译器根据需要自动调用 C run time library，而不需在程序中特别注明。C run time library 可在 signed char 类型除法等多重运算中被调用。

编译选项

使用多重中断的情况下，在ICC740的编译选项中需要追加-h。

专栏 使用中断表达式堆栈进行多重中断的应用

如上图所示,中断函数 B 使用中断表达式堆栈的情况下，有以下几种方法。

- 中断函数 A 使用中断表达式堆栈期间，将中断设定为中断禁止状态。
- 中断函数 B 的运算部分用汇编语言描述，但不适用于实时程序。
- 中断函数 A 没有使用中断表达式堆栈的情况下，中断函数 B 使用中断表达式堆栈没有任何问题。

（中断函数 A 或 B 的任一方使用中断表达式堆栈时,中断表达式堆栈的内容将不会被改写。）

多重中断的定义

在cstartup.s31的后面是为使用多重中断而进行的设定。

```
;------------------------------------------------------------------
; Turning off 'interruptable ISRs':
; Do this if you need the extra byte(s)
;
; 1. Uncomment the define below
; 2. Assemble this file
; 3. Include the result in your linker command file:
;    -C cstartup.r31
;
; Variable '?IES_USAGE' and its initialization will no longer
; be included.
;------------------------------------------
;#define NO_INTERRUPTABLE_ISR
```

使用多重中断时，保留此行。

因为在 default 中已设定成「使用」,所以使用时,按原样保留这一行。不使用的情况下，删除最后一行开头的分号「;」,即从注释行转换成有效指令。

```
#ifndef NO_INTERRUPTABLE_ISR
;------------------------------------------------------------------;
;  ?IES_USAGE - Determines if the IES is setup and used.
;
;  This variable is used for interrupt functions when compiling
;  with the '-h' option.
;------------------------------------------------------------------;

    RSEG    ZPAGE
    PUBLIC  ?IES_USAGE
?IES_USAGE:
    BLKB    1
#endif
```

```
#ifndef NO_INTERRUPTABLE_ISR
;------------------------------------------------------------------;
;  Initialize ?IES_USAGE:
;   1   IES not used
;   0   First use of IES, need to setup IES
;  <0   IES already setup and used
;------------------------------------------------------------------;
        LDA  #1
        STA  zp:?IES_USAGE
#endif
```

这两处是为多重中断而进行的设定。使用多重中断(NO_INTERRUPTABLE_ISR 没有被定义)的情况下,这一部分将被汇编。不使用(NO_INTERRUPTABLE_ISR 被定义)的情况下，这部分不被汇编。

多重中断处理函数的描述例

结合上述的注意事项,使用下述规范的程序例如下所示:

函数	处理内容	中断优先级	中断状态
INT1 中断处理函数	异常处理(紧急停止)	1	禁止多重处理
串行 I/O1 接收中断处理函数	通信接收	2	只允许一重多重处理
定时器 2 中断处理函数	一般处理	3	允许多重中断处理

在 main 函数③,只对希望产生中断的函数发许可位。在定时器 2 的中断处理函数的开头部分执行 CLI 命令可以向所有中断赋与许可, 因而可以执行多重中断。但是, 对定时器 2 自身的中断却不允许。串行 I/O1 接收中断处理函数中, INT1 以外的中断被设定成禁止, 通过执行 CLI 命令使得只有 INT1 中断能成为多重中断。
INT1 中断处理函数在函数执行过程中处于中断禁止状态, 所以不会发生多重中断。

另外, 在复位后也处于中断禁止状态 (I 标志=1)。进入中断函数, 中断则进入禁止状态(I 标志=1), 出了中断函数(RTI 命令),中断则进入许可状态(I 标志=0)。

```
#include    <intr740.h>      /* inline 函数用头文件 */
#include    "sfr_3803h.h"    /* 3803H group 用 SFR 头文件 */

void main( void )
{
        .
        .
        .
    /* ①设定中断边缘选择位(中断要素位) */
    INTEDGE.1 = 1 ;          /* INT1 上升沿有效沿*/

    /* ②在执行等待操作后,对应的中断请求位清 0(无请求) */
    nop_instruction() ;      /* 等待操作 */
    IREQ1.1 = 0 ;            /* INT1 中断请求位→ 清除 */
    IREQ1.2 = 0 ;            /* 串行 I/O1 接受中断请求位 →清除 */
    IREQ1.7 = 0 ;            /* 定时器 2 中断请求位 →清除 */

    /* ③所对应的中断允许位置 1(允许) */
    ICON1.1 = 1 ;            /* INT1 中断允许位 → 允许 */
    ICON1.2 = 1 ;            /* 串行 I/O1 接受中断允许位 → 允许 */
    ICON1.7 = 1 ;            /* 定时器 2 中断允许位 → 允许 */

    enable_interrupt() ;     /* 中断允许 CLI 指令 */

    while( 1 ) ;             /* 中断等待处理 */
}
```

只对希望产生中断的位
设置许可

```
#include    <intr740.h>     /* inline 函数用头文件 */
#include    "sfr_3803h.h"   /* 3803H group 用 SFR 头文件 */

unsigned char    Cntr ;
unsigned char    T_5msec = 1 ;
unsigned char    T_flg = 0 ;
unsigned char    Val ;

interrupt [28] void Int1( void )        /* INT1 中断处理函数[紧急停止] */
{
    Cntr = 0 ;
}

interrupt [26] void SIO1R( void )       /* 串行 I/O1 接收中断处理函数 */
{
    ICON1.2 = 0 ;           /* 串行 I/O1 接收中断允许位→ 禁止*/
    ICON1.7 = 0 ;           /* 定时器 2 中断允许位 → 禁止 */
    enable_interrupt() ;    /* 中断允许 CLI 指令 */
        ¦
    T_flg = 1 ;
        ¦
    disable_interrupt() ;   /* 中断禁止 SEI 指令 */
    ICON1.2 = 1 ;           /* 串行 I/O1 接收中断允许位 →允许 */
    ICON1.7 = 1 ;           /* 定时器 2 中断允许位 → 允许 */
}

interrupt [16] void Timer2( void )      /* 定时器 2 中断处理函数 */
{
    ICON1.7 = 0 ;           /* 定时器 2 中断允许位 → 禁止 */
    enable_interrupt() ;    /* 中断允许 CLI 命令 */

    if( T_5msec ){
        T_5msec = 0 ;
        if( Cntr < 9 ){
            Cntr++ ;
        } else {
            Cntr = 0 ;
        }
        if( T_flg ){
            Val = Cntr ;
            T_flg = 0 ;
        }
    } else {
        T_5msec = 1 ;
    }

    disable_interrupt() ;   /* 中断禁止 SEI 指令*/
    ICON1.7 = 1 ;           /* 定时器 2 中断允许位 → 允许 */
}
```

・多重中断禁止
・中断表达式堆栈
使用禁止
(参照注意事项)

・只有紧急停止时可产生多重中断
・中断表达式堆栈
使用禁止
(参照注意事项)

・多重中断可能
・中断表达式堆栈
可以使用

对定时器 2 本身的中断禁止

对定时器 2 自身的中断允许

参考文件

740 族用 C 汇编器组装文件
新闻
工具新闻
740 族的各种 MCU 使用说明书
技术信息
(最新版请从瑞萨的网页上下载。)

瑞萨网页

http://japan.renesas.com/

联系窗口
http://japan.renesas.com/inquiry
csc@renesas.com

修改记录	740 族编程指南<C 语言篇> 应用说明

Rev.	发行日	修改内容	
		页数	要点
1.00	2004.03.31	-	初版发行
2.00	2006.3.15	-	对文章中的措词和表达方法进行了全面修改。
		5	・C 语言的特长:(2)的文章表达方式有部分修改。 ・C 语言与汇编语言比较:表中的内容有部分修改。
		6	・M3T-ICC740 表中改成现有的产品名称。(4 处)
		8	<C 言語的规则> ・6 项→5 项 ・「原则上用小写字母描述程序。」删除 ・修改号码。
			<C 语言的源文件构成> ・内部变量的声明→内部变量的定义
		11	・※追加。(6 处)
		15	・标题变量的声明→变量的声明和定义 ・文章中 变量的声明→变量的声明和定义 ・文章中 声明→定义 ・文章中 声明→声明・定义
		16	・<1.2.3 数据的特性>中文章的一部分被删除。 ・(signed/unsigned 修饰符)的「修饰符」删除。 ・<明确表示带符号或不带符号>的第一行 类型修饰符→类型
		17	・表的内容修改。
		24	・<1.3.5 其他的运算符>中运算符的种类由 4 种→6 种
		25	<指针运算符> ・文中 表示→指定 ・表中 表示→指定
		27	・<「隐含类型转换」的解释错误及处理方法> 表中内容作了全面修改。
		33	・专栏题目有改动。
		39	图形中点线删除
		40	・「・・・」→「・・・・・・・・・・」 (3 处)
		42	・<1.5.3 函数间的数据传送>第三行有改动
		43	・<1.6.1 变量与函数的有效范围>文章有改动。 <变量与函数的有效范围> ・ 相关内容变动。 ・ 图中文章表现有改动。
		44	・<记忆级指定符>的文章删除了一部分。 ・<内部变量记忆级>的文章有一部分修改。
		46	・<1.6.3 函数的记忆级>文章有修改。 <整体函数和局部函数> ・ (2)与(3)的内容,结合以下源文件的内容交换。 ・ 表示不能调用的箭头的颜色改为红色。
		47	・<函数记忆级>的表中,extern 一拦中 调用一方→声明一方
		48	・<1.7.1 数组>文章有改动。
		50	<2 次元数组> ・ 第一行 「列」与「行」交换。 ・「2 维数组的概念」图中第二行与第三列删除。 ・「2 维数组的声明与初始化」下方 buff2[2][3]→buff2[][3]

修改记录	740 族编程指南<C 语言篇> 应用说明

Rev.	发行日	修改内容	
		页数	要点
2.00	2006.3.15	51	・<1.7.3 指针>中文章有改动。
			・<指针变量的声明>示意图有改动。
		52	・<指针与变量的关系>的文章有改动。
			<指针变量的运算>
			・文章中第 3 行:地址值→地址计算
			・第 7 行 ptr + 2→ptr + 2 × sizeof(int)
		53	<指针变量与 1 维数组>
			・图中指示线的颜色改变(2 处)。
			・"str"→str
			・ <指针变量与 2 维数组>的图有改动。
		54	<函数的地址交付>
			・文章有改动。
			・图形有改动。
		55	・图形有改动。
		56	<指针数组与 2 维数组>
			・删除一部分文章。
			・<2 维数组><指针数组>的程序例及图形删除一部分。
		57	・图形有改动。
		58	<1.8.1 结构体与共用体>的文章有一部分删除。
			・<从基本数据类型向结构体>中，文字表达有改动
		59	・小标题有改动，结构体的定义与宣言→结构体的定义
			・<结构体的定义>中,文章有改动。
		60	・图形有改动。
		62	・专栏中,一部分文章删除与程序例中的错误修正。
		63	・<1.9.1 ICC740 的预处理指令有改动
		64	・<1.9.2 读取文件>文章有改动。
		65	・<1.9.3 宏定义>表达修改。
		66	・<定义的取消>第 2 行 4 个→8 个
		67	・条件式→常数式 (2 处)
			・ 识别符→宏名称 (4 处)
		70	・<2.1.1 数码/数据的种类>文字表达有改动
			<ICC740 所形成的数据/代码>
			・自动变量→参数与自动变量
			・CSTR, CONST 段→CONST, CSTR 段
		71	・CODE 一栏中 程序→程序代码
		72	・<段的内存分配>最下行内容追加。
			・<Large 模型与 Tiny 模型>追加。
		73	・<强制向 ROM 段配置>const 数据类型 变量名称 ;→ const 数据类型 变量名称 = 值 ;
		74	・题目中「(#pragma codeseg)」删除。
			・第 4 行目中文章有一部分修改。
			・模块→文件 (3 处)
		75	・<初始化设定文件的构成>中断向量・表→中断向量
		76	・<启动程序的变更>中断向量的登录→中断向量区域尺寸的设定
		77	・ <中断向量的登录>改为<中断向量区域尺寸的设定、复位向量的登录>, 内容有全面修改。
		78	・<复位向量的登录>一项删除。

修改记录	740 族编程指南<C 语言篇> 应用说明

Rev.	发行日	改訂内容	
		页数	要点
2.00	2005.3.15	79	・追加第 4 行。
			・图框中 BITVARS 段的内容有改动。
		80	・第 3～4 行,文章表现有改动。
			・程序例的 INT_EXPR_STACK 部分修改成现在的 lnk740l.xcl 内容。
			・INT_EXPR_STACK 段的内容说明有改动。
			・※后面内容有改动。
		81	・增加专栏说明。
			・<special page 的开头/结束地址的设定>内容说明有改动。
		82	・第一个※后面的文章有改动。
		84	・<指针变量>的图有修改。
			・<指针变量的 zpage 指定>图形有改动。
		85	・(1)的文章有改动。(2 处)
		86	・程序例 short→unsigned char (6 处)
		87	・<SFR 区域的定义>head file 名 io3803.h→sfr_3803h.h (2 处)
		88	・<2.3.4 无法用 C 语言描述时的对策>文章的第 1 行删除。
		89	・2.4.1 函数间的接口→2.4.1 函数间的调用。
			・<2.4.1 函数间的调用>中,文章有改动。
			・<函数的入口处理与出口处理>的第 1 行文章有改动。
		90	・<栈的种类>表中,增加「寄存器入栈」。
		95	・head file 名称 io3803.h→sfr_3803h.h (2 处)
			・中断处理函数的表达方法修改。
			・该中断允许位为 0(禁止)的部分删除。
		96	・ <基本中断处理函数的描述 (C 语言)>的内容有全面修改。
		96	・<使用中断处理的注意事项>追加。
		97	・增加专栏。
		98	・第 1 行增加「根据 CLI 指令」。
		99	・<中断向量区域的登录>内容有全面修改
		101-104	・ ・<2.5.6 多重中断的使用方法>追加。

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors.
   Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.

注意

本文只是参考译文，前页所载英文版“Cautions”具有正式效力。

请遵循安全第一进行电路设计

1. 虽然瑞萨科技尽力提高半导体产品的质量和可靠性，但是半导体产品也可能发生故障。半导体的故障可能导致人身伤害、火灾事故以及财产损害。在电路设计时，请充分考虑安全性，采用合适的如冗余设计、利用非易燃材料以及故障或者事故防止等的安全设计方法。

关于利用本资料时的注意事项

1. 本资料是为了让用户根据用途选择合适的瑞萨科技产品的参考资料，不转让属于瑞萨科技或者第三者所有的知识产权和其它权利的许可。
2. 对于因使用本资料所记载的产品数据、图、表、程序、算法以及其它应用电路的例子而引起的损害或者对第三者的权力的侵犯，瑞萨科技不承担责任。
3. 本资料所记载的产品数据、图、表、程序、算法以及其它所有信息均为本资料发行时的信息，由于改进产品或者其它原因，本资料记载的信息可能变动，恕不另行通知。在购买本资料所记载的产品时，请预先向瑞萨科技或者经授权的瑞萨科技产品经销商确认最新信息。
   本资料所记载的信息可能存在技术不准确或者印刷错误。因这些错误而引起的损害、责任问题或者其它损失，瑞萨科技不承担责任。
   同时也请通过各种方式注意瑞萨科技公布的信息，包括瑞萨科技半导体网站。
   （http：//www.renesas.com）
4. 在使用本资料所记载部分或者全部数据、图、表、程序以及算法等信息时，在最终做出有关信息和产品是否适用的判断前，务必对作为整个系统的所有信息进行评价。由于本资料所记载的信息而引起的损害、责任问题或者其它损失，瑞萨科技不承担责任。
5. 瑞萨科技的半导体产品不是为在可能和人命相关的环境下使用的设备或者系统而设计和制造的产品。在研讨将本资料所记载的产品用于运输、交通车辆、医疗、航空宇宙用、原子能控制、海底中继器的设备或者系统等特殊用途时，请与瑞萨科技或者经授权的瑞萨产品经销商联系。
6. 未经瑞萨科技的书面许可，不得翻印或者复制全部或者部分资料的内容。
7. 如果本资料所记载的某产品或者技术内容受日本出口管理限制，必须在得到日本政府的有关部门许可后才能出口，并且不准进口到批准目的地国家以外的国家。
   禁止违反日本和（或者）目的地国家的出口管理法和法规的任何转卖、挪用或者再出口。
8. 如果需要了解本资料所记载的信息或者产品的详细，请与瑞萨科技联系。