740 Family

Programming Guidelines <C Language>

Preface

This application note is written for the Renesas 740 family 8-bit single-chip microcomputers.
It explains the basics of C language programming and how to put your program into ROM using the M3T–ICC740 C compiler.
For details about hardware and development support tools available for each type of microcomputer in the740 family, please refer to the appropriate hardware manuals, user's manuals and instruction manuals.

Guide to Using This Application Note

This application note provides programming guidelines for M3T-ICC74, the C compiler for the 740 family. Knowledge of 740 family microcomputer architectures and the assembly language is helpful in using this manual. The manual contains the following:

- Those who learn the C language for the first time Begin with Chapter 1.
- Those who wish to know ICC740 extended functions Begin with Chapter 2.

This application note is described with the case of the large memory model (-ml option). Please refer to ICC compiler programming guide (icc740_jp.pdf) about options.

Before using material, please visit our website to confirm that this is the most current document available.

Table of Contents

Chapter 1
Introduction to C Language

This chapter explains for those who learn the C language for the first time the basics of the C language that are required when creating a built-in program.

1.1 Programming in C Language

1.1.1 Assembly Language and C Language

The following explains the main features of the C language and describes how to write a program in "C".

Features of the C Language

(1) An easily traceable program can be written.
The basics of structured programming, i.e., "sequential processing", "branch processing", and "repeat processing", can all be written in a control statement. For this reason, it is possible to write a program whose flow of processing can easily be traced.

(2) A program can easily be divided into modules.
A program written in the C language consists of basic units called "functions". Since function have their parameters highly independent of others, a program can easily be made into parts and can easily be reused. Furthermore, modules written in the assembly language can be incorporated into a C language program directly without modification.

(3) An easily maintainable program can be written.
For reasons (1) and (2) above, the program after being put into operation can easily be maintained. Furthermore, since the C language is based on standard specifications (ANSI standard (Note)), a program written in the C language can be ported into other types of microcomputers after only a minor modification of the source program.

Note: This refers to standard specifications stipulated for the C language by the American National Standards Institute (ANSI) to maintain the portability of C language programs.

Comparison between C and Assembly Languages

The folllowing outlines the differences between the C and assembly languages with respect to the method for writing a source program.

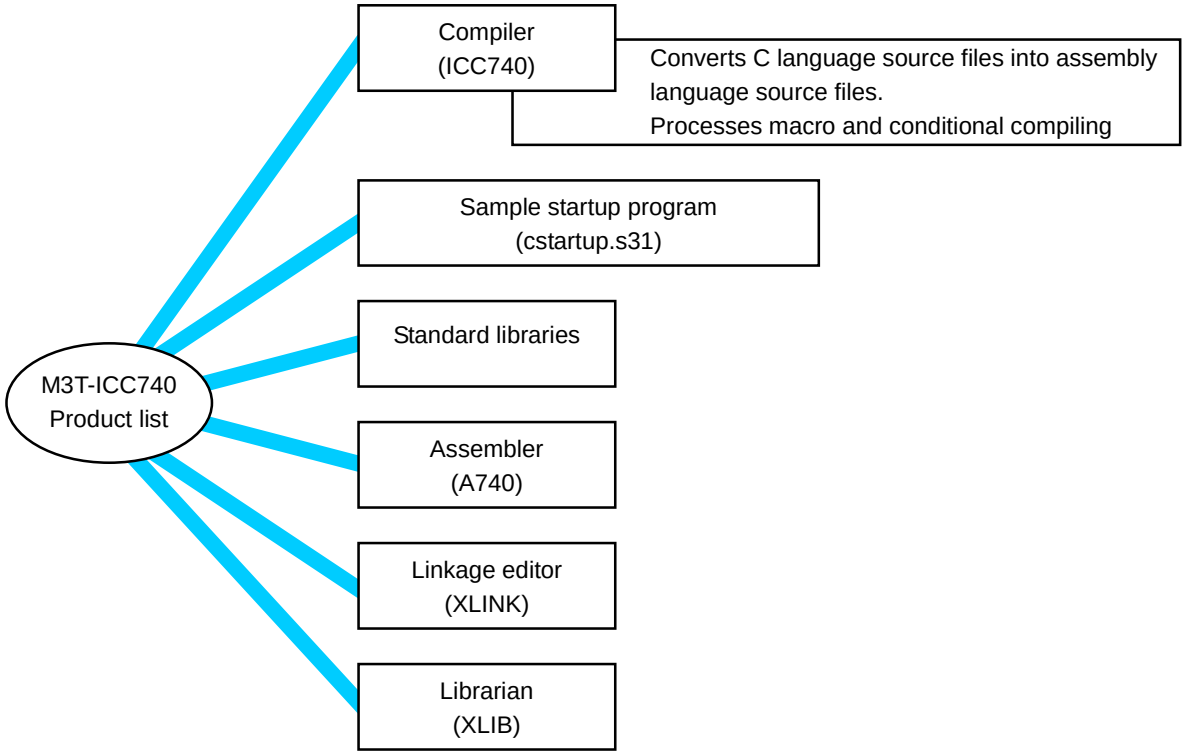
	C language	Assembly language
Basic unit of program (Method of description)	Function (Function name ( ) { })	Subroutine (Subroutine name:)
Format	Free format	1 instruction/line
Discrimination between uppercase and lowercase	Uppercase and lowercase are discriminated (Normally written in lowercase)	Not discriminated
Allocation of data area	Specified by "data type"	specified by a number of bytes (using pseudo-instruction)

1.1.2 Program Development Procedure

The operation of translating a source program written in "C" into machine language is referred to as "compiling". The software provided for performing this operation is called a "compiler". This section explains the procedure for developing a program by using M3T-ICC740, the C compiler for the 740 family of Renesas 8-bit single-chip microcomputers.

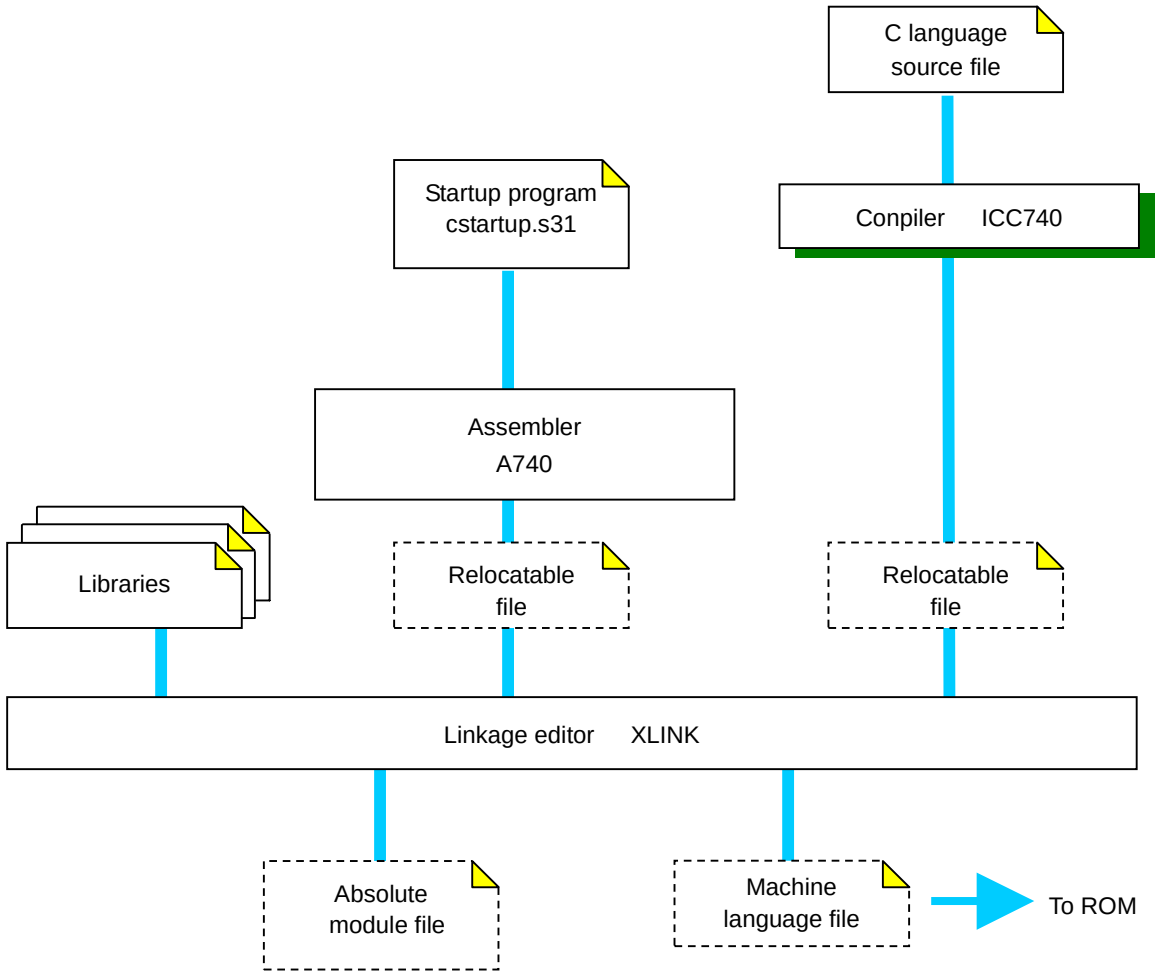
M3T-ICC740 Product List

The following lists the products included in M3T-ICC740, the C compiler for the Renesas 8-bit single-chip microcomputers 740 family.



Creating Machine Language File from Source File

Creation of a machine language file requires the conversion of start-up programs written in Assembly language and C language source files.
The following shows the tool chain necessary to create a machine language file from a C language source file.



1.1.3 Easily Understandable Program

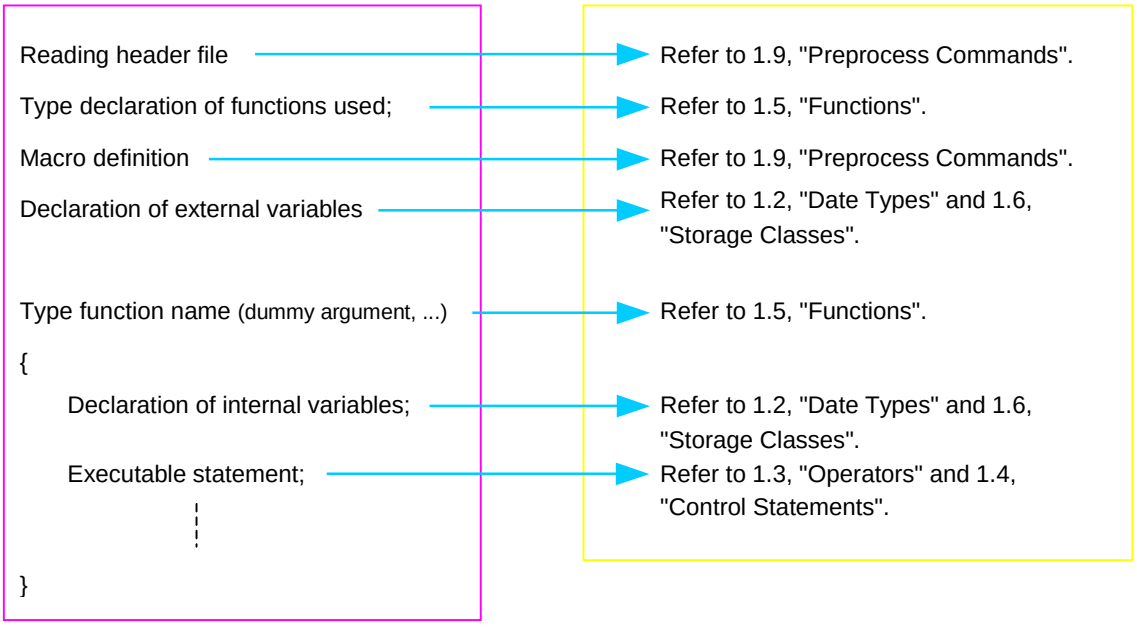
Since there is no specific format for C language programs, they can be written in any desired way only providing that some rules stipulated for the C language are followed. However, a program must be easily readable and must be easy to maintain. Therefore, a program must be written in such a way that everyone, not just the one who developed the program, can understand it.
This section explains some points to be noted when writing an "easily understandable" program.

Rules on C Language

The following lists the six items that need to be observed when writing a C language program:
(1) As a rule, use lowercase English letters to write a program.
(2) Separate executable statements in a program with a semicolon ";".
(3) Enclose execution units of functions or control statements with brackets "{" and "}"
(4) Functions and variables require type declaration.
(5) Reserved words cannot be used in identifiers (e.g., function names and variable names).
(6) The comment is described with "/* comment */" or "//comment" (C++ form)."-K" option is required in the case of C++ form.

Configuration of C Language Source File

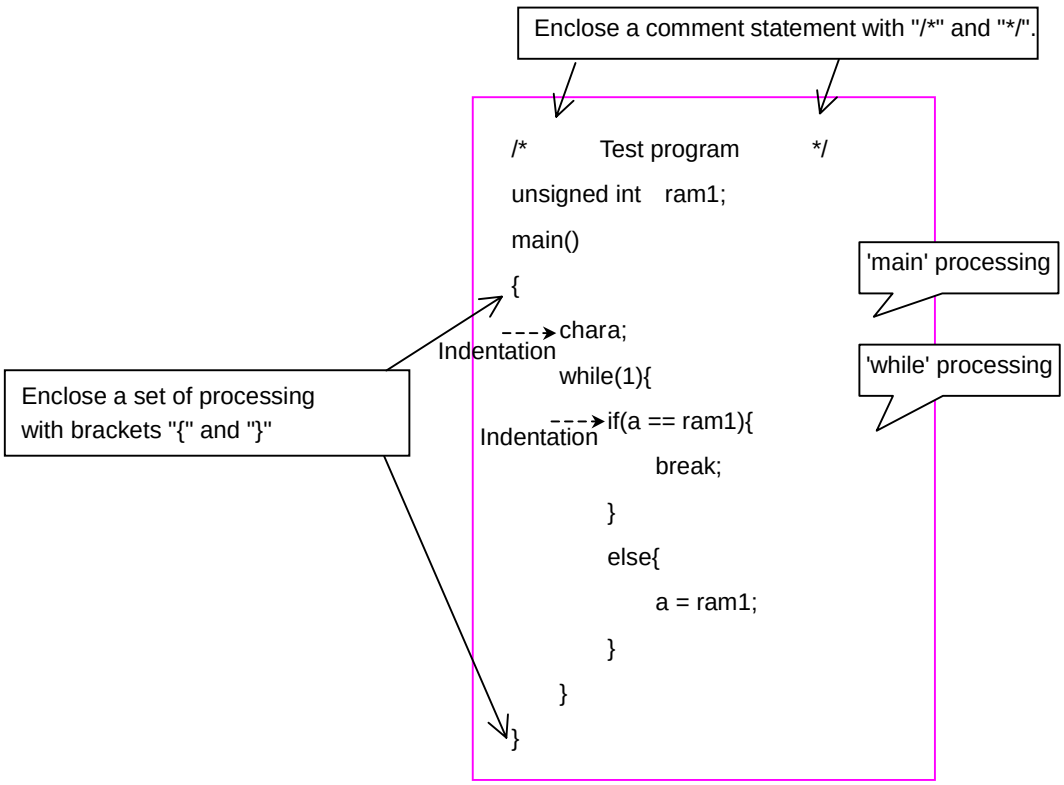
The following schematically shows a configuration of a general C language source file. For each item in this file, refer to the section indicated with an arrow.



Programming Style

To improve program maintainability, programming conversions should be agreed upon by the programming team. Creating a template is a good way for the developers to establish a common programming style that will facilitate program development, debug and maintenance. The following shows an example of a programming style.

(1) Create separate functions for various tasks of a program.
(2) Keep functions relatively small (< 50 lines is recommended)
(3) Do not write multiple executable statements in one line
(4) Indent each processing block successively (normally 4 tab stops)
(5) Clarify the program flow by writing comment statements as appropriate
(6) When creating a program from multiple source files, place the common part of the program in an independent separate file and share it



Method for Writing Comments

Comments are an important aspect of a well written program. Program flow can be clarified, for example, through a file and function headers.

Example of file header

```
/* ""FILE COMMENT"" ************************************************************
   *  System Name      : Test program
   *  File Name        : TEST.C
   *  Version          : 1.00
   *  Contents         : Test program
   *  Customer         :……
   *  Model            :……
   *  Order            :……
   *  CPU              : M38039MC-XXXFP
   *  Compiler         : M3T-ICC740 (Ver.1.00)
   *  Programmer       : XXXX
   *  Note             :The module contained in this file is designed so that it can be reused.
   ******************************************************************************
   *  Copyright,XXXX xxxxxxxxxxxxxxxx CORPORATION
   ******************************************************************************
   *  History          :XXXX.XX.XX          : Start
   *  ""FILE COMMENT END"" ********************************************************************/

   /* ""Prototype declaration"" ****************************************************************/
   void     main (void);
   void     key_in (void);
   void     key_out (void);
```

Example of function header

```
   /* ""FUNC COMMENT"" ************************************************************
   *  ID               : 1.
   *  Module outline   : Main function
   *  ---------------------------------------------
   *  Include          : "system.h"
   *  ---------------------------------------------
   *  Declaration      : void main (void)
   *  ---------------------------------------------
   *  Functionality    : Overall controll
   *  ---------------------------------------------
   *  Argument         : void
   *  ---------------------------------------------
   *  Return value     : void
   *  ---------------------------------------------
   *  input            : None
   *  Output           : None
   *  ---------------------------------------------
   *  Used functions   : void   key_in (void)        : Input function
   *                     : void   key_out (void)     : Output function
   *  ---------------------------------------------
   *  Precaution       : Nothing particular
   *  ---------------------------------------------
   *  History          : XXXX.XX.XX                  : Start
   /* ""FUNC COMMENT END"" ********************************************************************/
   #include     "system.h"
   void   main (void)
   {
      while(1){          /*    Endless loop   */
            key_in();    /*    Input processing    */
            key_out();   /*    Output processing   */
      }
   }
```

Column Reserved Words of ICC740

The words listed in the following are reserved for ICC740. Therefore, these words cannot be used in variable or function names.

__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	

* When using "-e" option, this word is reserved for ICC740.

1.2 Data Types

1.2.1 "Constants" in C Language

Four types of constants can be handled in the C language: "integer", "real", "single character" and "character string".
This section explains the method of description and the precautions to be noted when using each of these constants.

Integer Constants

Integer constants can be written using one of three methods of numeric representation: decimal, hexadecimal, and octal. The following shows each method for writing integer constants. Constant data are not discriminated between uppercase and lowercase.

Numeration	Method of writing	Examples
Decimal	Normal mathematical notation (nothing added)	127, +127, -56
Hexadecimal	Numerals are preceded by 0x or 0X	0x3b, 0x3B
Octal	Numerals are preceded by 0 (zero)	07, 041

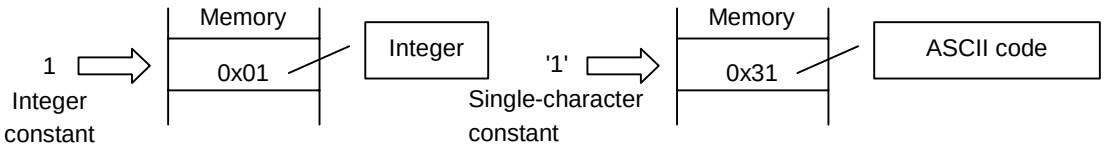
Real Constants (Floating-point Constants)

Floating-point constants refer to signed real numbers that are expressed in decimal. These numbers can be written by usual method of writing using the decimal point or by exponential notation using "e" or "E".

- Usual method of writing Example: 175.5, -0.007
- Exponential notation Example: 1.755e2, -7.0E-3

Single-character Constants

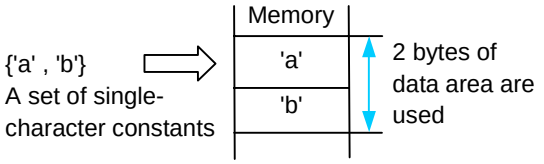
Single-character constants must be enclosed with single quotations ('). In addition to alphanumeric characters, control codes can be handled as single-character constants.
Inside the microcomputer, all of these constants are handled as ASCII code, as shown below.

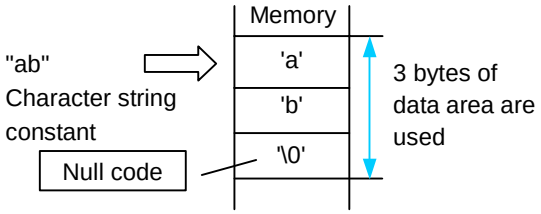


Character String Constants

A row of alphanumeric characters or control codes enclosed with double quotations (") can be handled as a character string constant. Character string constants have the null character "\0" automatically added at the end of data to denote the end of the character string.

Example: "abc", "012\n", "Hello!"





Column List of Control Codes (Escape Sequence)

The following shows control codes (escape sequence) that are frequently used in the C language.

Notation	Contents
\f	Form feed (FF)
\n	New line (NL)
\r	Carriage return (CR)
\t	Horizontal tab (HT)
\\	\symbol
\'	Single quotation
\"	Double quotation
\x constant value	Hexadecimal
\ constant value	Octal
\0	Null code

1.2.2 Variables

Before a variable can be used in a C language program, its "data type" must first be declared in the program. The data type of a variable is determined based on the memory size allocated for the variable and the range of values handled.
This section explains the data types of variables that can be handled by ICC740 and how to declare the data types.

Basic Data Types of ICC740

The following lists the data types that can be handled in ICC740. Descriptions enclosed with ( ) in the table below can be omitted when declaring the data type.

	Data type	Bit length	Range of values that can be expressed
integer	char	8 bits	0 to 255
	unsigned char		0 to 255
	signed char		-128 to 127
	unsigned short (int)	16 bits	0 to 65535
	(signed) short (int)		-32768 to 32767
	unsigned int	16 bits	0 to 65535
	(signed) int		-32768 to 32767
	unsigned long (int)	32 bits	0 to 4294967295
	(signed) long (int)		-2147483648 to 2147483647
Real	float	32 bits	Number of significant digits: 9
	double	32 bits	Number of significant digits: 9
	long double	32 bits	Number of significant digits: 9

* When using "-c" option, the data range which can be expressed is –128 to 127 since a char type is equivalent to a signed char type.

Declaration of Variables

Variables are declared using a format that consists of a "data type variable name;".

Example: To declare a variable a as char type

```
char a;
```

By writing "data type variable name = initial value;", a variable can have its initial value set simultaneously when it is declared.

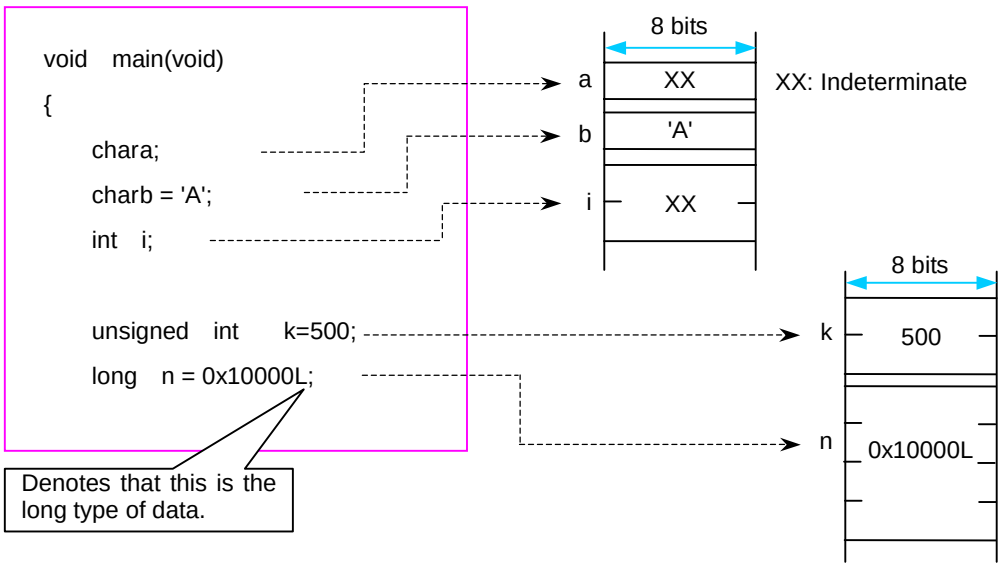
Example: To set 'A' to variable a of char type as its initial value

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

Furthermore, by separating an enumeration of multiple variables with a comma (,), variables of the same type can be declared simultaneously.

Example: int i, j;

Example: int i = 1, j = 2;



1.2.3 Data Characteristics

When declaring a variable or constant, ICC740 allows its data characteristic to be written along with the data type. The specifier used for this purpose is called the "type qualifier".
This section explains the data characteristics handled by ICC740 and how to specify a data characteristic.

Specifying that the Variable or Constant is Singed or Unsigned Data (singed/ unsigned Qualifier)

Write the type qualifier "signed" when the variable or constant to be declared is signed data or "unsigned" when it is unsigned data. If neither of these type specifiers is written when declaring a variable or constant, ICC740 assumes that it is signed data for only the data type char, or unsigned data for all other data types.

```
void    main(void)
{
        chara;
        signed    chars_a;

        int    b;
        unsigned    int      u_b;
                    :
}
```

Synonymous with "unsigned char a;"

Synonymous with "signed int b;"

* When using "-c" option, a char type is equivalent to a signed char type

Specifying that the Variable or Constant is Constant Data (const Qualifier)

Write the type qualifier "const" when the variable or constant to be declared is the data whose value does not change at all even when the program is executed. If a description is found in the program that causes this constant data to change, ICC740 outputs an error.

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

        a = 5;
        c_a = 5;
}
```

Error is generated

Inhibiting Optimization by Compiler (volatile Qualifier)

ICC740 optimizes the instructions that do not have any effect in program processing, thus preventing unnecessary instruction code from being generated. However, there are some data that are changed by an interrupt or input from a port irrespective of program processing. Write the type qualifier "volatile" when declaring such data. ICC740 does not optimize the data that is accompanied by this type qualifier and outputs instruction code for it.

```
char port1;
volatile char port2;

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

Because the qualifier "volatile" is nonexistent in the data declaration, comparison is removed by optimization and no code is output for this.

Because the qualifier "volatile" is specified in the data declaration, no optimization is performed and code is output for this.

Column Syntax of Declaration

When declaring data, write data characteristics using various specifiers or qualifiers along with the data type. The following shows the syntax of a declaration.

Declaration specifier			Declarator (data name)
Storage class specifier (described later)	Type qualifier	Type specifier	
static register auto extern	unsigned signed const volatile	int char float struct union	data name

1.3 Operators

1.3.1 Operators of ICC740

ICC740 has various operators available for writing a program.
This section describes how to use these operators for each specific purpose of use (not including address and pointer operators) and the precautions to be noted when using them.

ICC740 Operators

The following lists the operators that can be used in ICC740.

Monadic arithmetic operators	++ -- + -
Binary arithmetic operators	+ -* / %
Shift operators	<< >>
Bitwise operators	& | ^ ~
Relation operators	> < >= <= == !=
Logical operators	&& || !
Assignment operators	= += -= *= /= %= <<= >>= &= |= ^=
Conditional operators	?:
sizeof operators	sizeof
Cast operators	(type)
Address operators	&
Pointer operators	*
Comma operators	,

1.3.2 Operators for Numeric Calculations

The primary operators used for numeric calculations consist of the "arithmetic operators" to perform calculations and the "assignment operators" to store the results in memory.
This section explains these arithmetic and assignment operators.

Monadic Arithmetic Operators

Monadic arithmetic operators return one answer for one variable.

Operator	Description format	Content
++	++ variable (prefix type) variable ++ (postfix type)	Increments the value of an expression.
--	-- variable (prefix type) variable -- (postfix type)	Decrements the value of an expression.
+	+ expression	Returns the value of an expression.
-	- expression	Returns the value of an expression after inverting its sign.

When using the increment operator (++) or decrement operator (--) in combination with an assignment or relational operator, note that the result of operation may vary depending on which type, prefix or postfix, is used when writing the operator.
<Examples>
Prefix type: The value is increment or decrement before assignment.
b = ++a; → a = a + 1; b = a;
Postfix type: The value is increment or decrement after assignment.
b = a++; → b = a; a = a + 1;

Binary Arithmetic Operators

In addition to ordinary arithmetic operations, these operators make it possible to obtain the remainder of an "integer divided by integer" operation.

Operator	Description format	Content
+	Expression 1 + expression 2	Returns the sum of expression 1 and expression 2 after adding their values
-	Expression 1 - expression 2	Returns the difference between expression 1 and expression 2 after subtracting their values
*	Expression 1 * expression 2	Returns the product of expression 1 and expression 2 after multiplying their values
/	Expression 1 / expression 2	Returns the quotient of expression 1 after dividing its value by that of expression 2
%	Expression 1 % expression 2	Returns the remainder of expression 1 after dividing its value by that of expression 2

Assignment Operators

The operation of "expression 1 = expression 2" assigns the value of expression 2 for expression 1. The assignment operator '=' can be used in combination with arithmetic operators described above or bitwise or shift operators that will be described later. (This is called a compound assignment operator.) In this case, the assignment operator '=' must always be written on the right side of the equation.

Operator	Description format	Content
=	expression 1 = expression 2	Substitutes the value of expression 2 for expression 1.
+=	expression 1 += expression 2	Adds the values of expressions 1 and 2, and substitutes the sum for expression 1.
-=	expression 1 -= expression 2	Subtracts the value of expression 2 from that of expression 1, and substitutes the difference for expression 1.
*=	expression 1 *= expression 2	Multiplies the values of expressions 1 and 2, and substitutes the product for expression 1.
/=	expression 1 /= expression 2	Divides the value of expression 1 by that of expression 2, and substitutes the quotient for expression 1.
%=	expression 1 %= expression 2	Divides the value of expression 1 by that of expression 2, and substitutes the remainder for expression 1.
<<=	expression 1 <<= expression 2	Shifts the value of expression 1 left by the amount equal to the value of expression 2, and substitutes the result for expression 1.
>>=	expression 1 >>= expression 2	Shifts the value of expression 1 right by the amount equal to the value of expression 2, and substitutes the result for expression 1.
&=	expression 1 &= expression 2	ANDs the bits representing the values of expressions 1 and 2, and substitutes the result for expression 1.
|=	expression 1 |= expression 2	ORs the bits representing the values of expressions 1 and 2, and substitutes the result for expression 1.
^=	expression 1 ^= expression 2	XORs the bits representing the values of expressions 1 and 2, and substitutes the result for expression 1.

Column Implicit Type Conversion

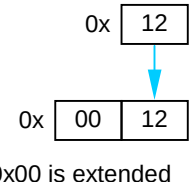
When performing arithmetic or logic operation on different types of data, ICC740 converts the data types following the rules shown below. This is called "implicit type conversion".

- Data types are adjusted to the data type whose bit length is greater than the other before performing operation.
- When substituting, data types are adjusted to the data type located on the left side of the equation.

When ...

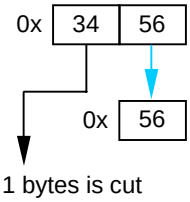
```
char    byte = 0x12;
int     word = 0x3456;
```

word = byte;
/* int ← char */



0x00 is extended

byte = word;
/* char ← int */



Upper 1 bytes is cut

1.3.3 Operators for Processing Data

The operators frequently used to process data are "bitwise operators" and "shift operators". This section explains these bitwise and shift operators.

Bitwise Operators

Use of bitwise operators makes it possible to mask data and perform active conversion.

Operator	Description format	Content
&	expression 1 & expression 2	Returns the logical product of the values of expressions 1 and 2 after ANDing each bit.
|	expression 1 | expression 2	Returns the logical sum of the values of expressions 1 and 2 after ORing each bit.
^	expression 1 ^ expression 2	Returns the exclusive logical sum of the values of expressions 1 and 2 after XORing each bit.
~	~expression	Returns the value of the expression after inverting its bits.

Shift Operators

In addition to shift operation, shift operators can be used in simple multiply and divide operations. (For details, refer to "Column Multiply and divide operations using shift operators".)

Operator	Description format	Content
<<	expression 1 << expression 2	Shifts the value of expression 1 left by the amount equal to the value of expression 2, and returns the result.
>>	expression 1 >> expression 2	Shifts the value of expression 1 right by the amount equal to the value of expression 2, and returns the result.

Comparison between Arithmetic and Logical Shifts

When executing "shift right", note that the shift operation varies depending on whether the data to be operated on is singed or unsigned.

- When unsigned → Logical shift: A logic 0 is inserted into the most significant bit.
- When signed → Arithmetic shift: Shift operation is performed so as to retain the sign. Namely, if the data is a positive number, a logic 0 is inserted into the most significant bit; if a negative number, a logic 1 is inserted into the most significant bit.

	<Unsigned> unsigned int i = 0xFC18 (i= 64520)	<Negative number> signed int i = 0xFC18 (i= -1000)		<Positive number> 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)
	Logical shift	Arithmetic shift (positive or negative sign is retained)			

Column Multiply and Divide Operations Using Shift Operators

Shift operators can be used to perform simple multiply and divide operations. In this case, operations are performed faster than when using ordinary multiply or divide operators. Considering this advantage, ICC740 generates shift instructions, instead of multiply instructions, for such operations as "*2", "*4", and "*8".

- Multiplication: Shift operation is performed in combination with add operation.
  a*2 → a<<1
  a*4 → a<<2
  a*8 → a<<3

- Division: The data pushed out of the least significant bit makes it possible to know the remainder.
  a/4 → a>>2
  a/8 → a>>3
  a/16 → a>>4

1.3.4 Operators for Examining Condition

Used to examine a condition in a control statement are "relational operators" and "logical operators". Either operator returns a logic 1 when a condition is met and a logic 0 when a condition is not met.
This section explains these relational and logical operators.

Relational operators

These operators examine two expressions to see which is larger or smaller than the other. If the result is true, they return a logic 1; if false, they return a logic 0.

Operator	Description format	Content
<	expression 1 < expression 2	True if the value of expression 1 is smaller than that of expression 2; otherwise, false.
<=	expression 1 <= expression 2	True if the value of expression 1 is smaller than or equal to that of expression 2; otherwise, false.
>	expression 1 > expression 2	True if the value of expression 1 is larger than that of expression 2; otherwise, false.
>=	expression 1 >= expression 2	True if the value of expression 1 is larger than or equal to that of expression 2; otherwise, false.
==	expression 1 == expression 2	True if the value of expression 1 is equal to that of expression 2; otherwise, false.
|=	expression 1 |= expression 2	True if the value of expression 1 is not equal to that of expression 2; otherwise, false.

Logical operators

These operators are used along with relational operators to examine the combinatorial condition of multiple condition expressions.

Operator	Description format	Content
&&	expression 1 && expression 2	True if both expressions 1 and 2 are true; otherwise, false.
||	expression 1 || expression 2	False if both expressions 1 and 2 are false; otherwise, true.
!	!expression 2	False if the expression is true, or true if the expression is false.

1.3.5 Other Operators

This section explains four types of operators which are unique in the C language.

Conditional Operator

This operator executes expression 1 if a condition expression is true or expression 2 if the condition expression is false. If this operator is used when the condition expression and expressions 1 and 2 both are short in processing description, coding of conditional branches can be simplified. The following lists this conditional operator and an example for using this operator.

Operator	Description format	Content
? :	Condition expression ? expression 1 : expression 2	Executes expression 1 if the condition expression is true or expression 2 if the condition expression is false.

• Value whichever larger is selected.

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

=

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

• Absolute value is found.

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

=

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

sizeof Operator

Use this operator when it is necessary to know the number of memory bytes used by a given data type or expression.

Operator	Description format	Content
sizeof()	sizeof expression sizeof (data type)	Returns the amount of memory used by the expression or data type in units of bytes.

Cast Operator

When operation is performed on data whose types differ from each other, the data used in that operation are implicitly converted into the data type that is largest in the expression. However, since this could cause an unexpected fault, a cast operator is used to perform type conversions explicitly.

Operator	Description format	Content
( )	(new data type) variable	Converts the data type of the variable to the new data type.

Address Operator

The address value of memory area in which variables are assigned is returned. Variable parts can be an array element. In that case, the address of the position which an element number shows will become its value.

Operator	Description format	Content
&	& variable	Returns the address of variable.

Pointer Operator

The contents of the memory area specified by the pointer variable are indicated.

Operator	Description format	Content
*	* variable	The contents of the memory area specified by the pointer variable are indicated.

Comma (Sequencing) Operator

This operator executes expression 1 and expression 2 sequentially from left to right. This operator, therefore, is used when enumerating processing of short descriptions.

Operator	Description format	Content
,	expression 1, expression 2	Executes expression 1 and expression 2 sequentially from left to right.

1.3.6 Priorities of Operators

The operators used in the C language are subject to "priority resolution" and "rules of combination" as are the operators used in mathematics.
This section explains priorities of the operators and the rules of combination they must follow:

Priority Resolution and Rules of Combination

When multiple operators are included in one expression, operation is always performed in order of operator priorities beginning with the highest priority operator. When multiple operators of the same priority exist, the rules of combination specify which operator, left or right, be executed first.

Priority resolution	Type of operator	Operator	Rules of combination
High	Expression	() [] -> . (Note1)	→
↑	Monadic arithmetic operators, etc.	! ~ ++ -- + - * (Note 2) & (Note 3) (type) sizeof	←
	Multiply/divide operators	* (Note 4) / %	→
	Add/subtract operators	+ -	→
	Shift operator	<< >>	→
	Relational operator (comparison)	< <= > >=	→
	Relational operator (equivalent)	== !=	→
	Bitwise operator (AND)	&	→
	Bitwise operator (EOR)	^	→
	Bitwise operator (OR)	|	→
	Logical operator (AND)	&&	→
	Logical operator (OR)	||	→
	Conditional operator	?:	←
↓	Assignment operator	= += -= *= /= %= &= ^= |= <<= >>=	←
Low	Comma operator	,	→

Note 1: The dot '•' denotes a member operator that specifies struct and union members.
Note 2: The asterisk '*' denotes a pointer operator that indicates a pointer variable.
Note 3: The ampersand '&' denotes an address operator that indicates the address of a variable.
Note 4: The asterisk '*' denotes a multiply operator that indicates multiplication.

1.3.7 Examples for Easily Mistaken Use of Operators

The program may not operate as expected if the "implicit conversion" or "precedence" of operators are incorrectly interpreted.
This section shows examples for easily mistaken use of operators and how to correct.

Incorrectly Interpreted "Implicit Conversion" and How to Correct

When an operation is performed between different types of data in ICC740, the data types are adjusted to that of data which is long in bit length by what is called "implicit conversion" before performing the operation. To ensure that the program will operate as expected, write explicit type conversion using the cast operator.

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

The expected operation is such that the operation result 0xff of expression (a - 1) in the if statement is compared with the value of variable b, which should hold true, but actually is found to be false.

The constant is handled as signed quantity (signed int). For this reason, the expression (a - 1) becomes an expression unsigned char - signed int. By "implicit conversion," unsigned char has its type changed to signed int, so the expression (a - 1) is calculated as (signed int - signed int).
The comparison of the operation result of expression (a - 1) with the variable b also is performed in the form of (signed int >= signed int) as a result of "implicit conversion." All told, comparison is performed on (0x00ff >= 5), so the result is found to be false.

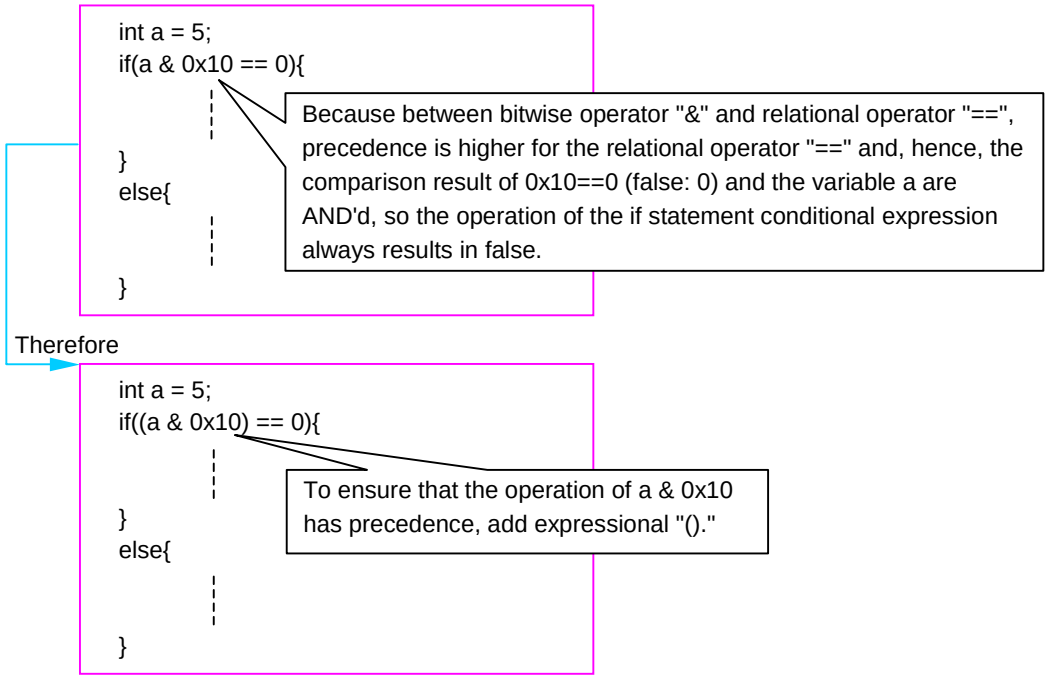
Therefore

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

Use the cast operator for the entire calculation result of expression (a - 1) to explicitly convert its type to unsigned char. In this way, its data type can be matched to that of variable b.

Incorrectly Interpreted "Precedence" of Operators and How to Correct

When one expression includes multiple operators, the "precedence" and "associativity" of operators need to be interpreted correctly. Also, to ensure that the program will operate as expected, use expressional "()."



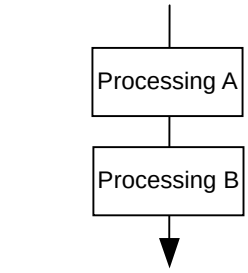
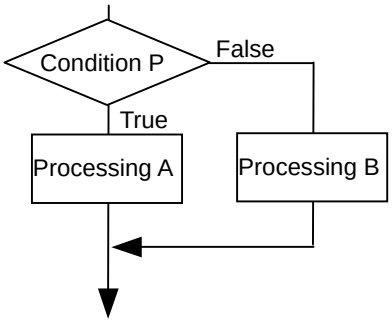
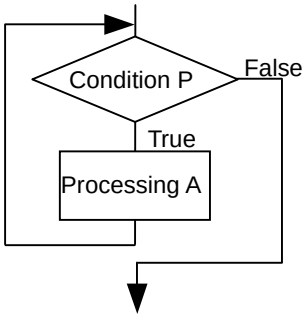
1.4 Control Statements

1.4.1 Structuring of Program

The C language allows "sequential processing", "branch processing" and "repeat processing"-- the basics of structured programming--to be written using control statements. Consequently, all programs written in the C language are structured. This is why the processing flow in C language programs are easy to understand.
This section describes how to write these control statements and shows some examples of usage.

Structuring of Program

The most important point in making a program easy to understand is to create a readable program flow. This requires preventing the program flow from being directed freely as one wishes. Therefore, processing flow is limited to the three primary forms: "sequential processing", "branch processing" and "repeat processing". The result is the technique known as "structured programming".
The following shows the three basic forms of structured programming.

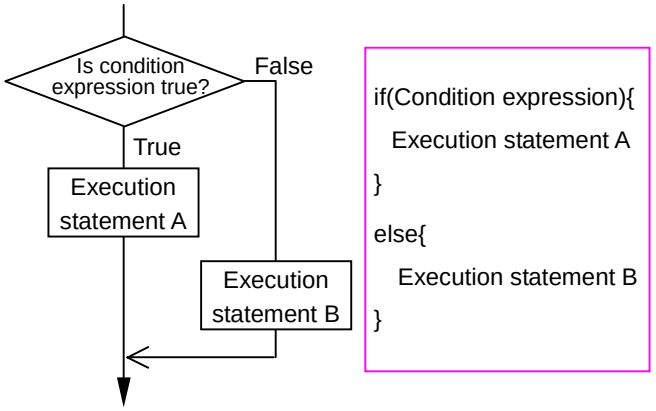
Sequential processing		Executed top down, from top to bottom.
Branch processing		Branched to processing A or processing B depending on whether condition P is true or false.
Repeat processing		Processing A is repeated as long as condition P is met.

1.4.2 Branching Processing Depending on Condition (Branch Processing)

Control statements used to write branch processing include "if-else", "else-if", and "switch-case" statements.
This section explains how to write these control statements and shows some examples of usage.

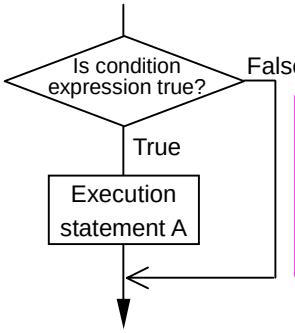
if-else Statement

This statement executes the next block if the given condition is true or the "else" block if the condition is false. Specification of an "else" block can be omitted.




```
if(Condition expression){
   Execution statement A
}
else{
   Execution statement B
}
```

• If the else statement is omitted




```
if(Condition expression){
Execution statement A
}
```

Count Up (if-else Statement)

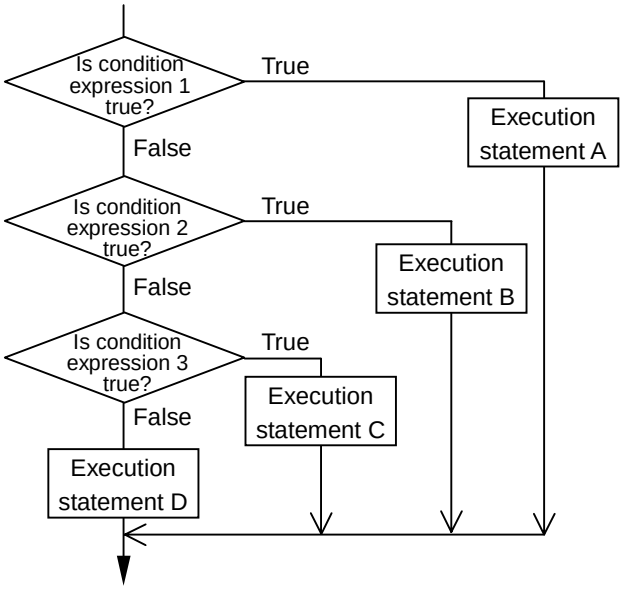
In this example, the program counts up a seconds counter "second" and a minutes counter "minute". When this program module is called up every 1 second, it functions as a clock.

```
void count_up(void);
unsigned int second = 0;
unsigned int minute = 0;

void count_up(void)
{
    if(second >= 59){
        second = 0;
        minute ++;
    }
    else{
        second ++;
    }
}
```

Declares "count_up" function. (Refer to Section 1.5, "Functions".)
Declares variables for "second" (seconds counter) and "minute" (minutes counter).
Defines "count_up" function.
If greater than 59 seconds, the module resets "second" and counts up "minute".
If less than 59 seconds, the module counts up "second".

else-if Statement

Use this statement when it is necessary to divide program flow into three or more flows of processing depending on multiple conditions. Write the processing that must be executed when each condition is true in the immediately following block. Write the processing that must be executed when none of conditions holds true in the last "else" block.




```
if(condition expression 1) {
        Execution statement A
}
else if(condition expression 2) {
        Execution statement A
}
else if(condition expression 3) {
        Execution statement A
}
else{
        Execution statement A
}
```

Switchover of Arithmetic Operations (else-if Statement)

In this example, the program switches over the operation to be executed depending on the content of the input data "sw".

```
void select(void);

int a = 29, b = 40;
long int ans;
char sw;

void select(void)
{
    if(sw == 0){
        ans = a + b;
    }
    else if(sw == 1){
        ans = a – b;
    }
    else if(sw == 2){
        ans = a * b;
    }
    else if(sw == 3){
        ans = a / b;
    }
    else{
        error();
    }
}
```

Declares "select" function.
(Refer to Section 1.5, "Functions".)

Declares the variables used.

Defines "select" function.

If the content of "sw" is 0,
the program adds data.

If the content of "sw" is 1,
the program subtracts data.

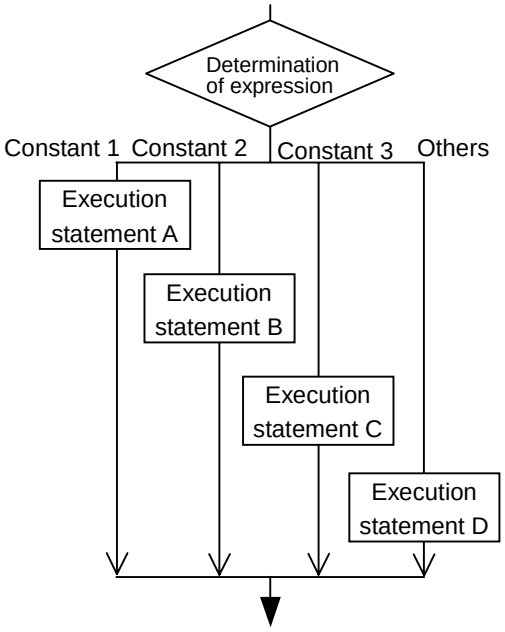
If the content of "sw" is 2,
the program multiplies data.

If the content of "sw" is 3,
the program devides data.

If the content of "sw" is 4 or greater,
the program performs error processing.

switch-case Statement

This statement causes program flow to branch to one of multiple processing depending on the result of a given expression. Since the result of an expression is handled as a constant when making decision, no relational operators, etc. can be used in this statement.




```
switch(expression) {

    case constant 1:   execution statement A
                       break;

    case constant 2:   execution statement B
                       break;

    case constant 3:   execution statement C
                       break;

    default:           execution statement D
                       break;

}
```

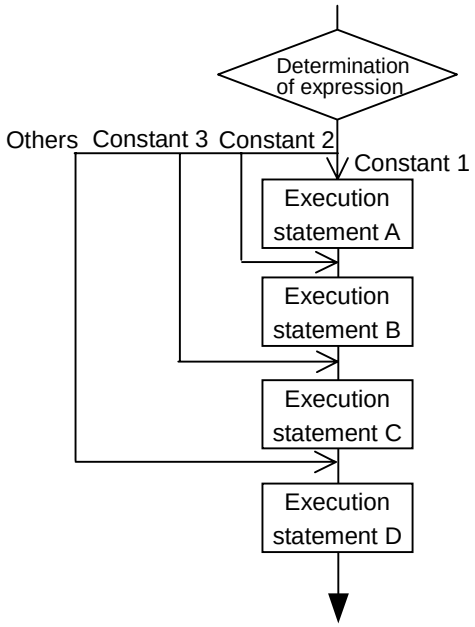
Switchover of Arithmetic Operations (switch-case Statement)

In this example, the program switches over the operation to be executed depending on the content of the input data "sw".

```
void select(void);                  <- Declares "select" function.
                                       (Refer to Section 1.5, "Functions".)
int a = 29, b = 40;                 <- Declares the variables used.
long int ans;
char sw;

void select(void)                   <- Defines "select" function.
{
    switch(sw){                     <- Deterrmines the content of "sw".

        case 0 : ans = a + b;       <- If the content of "sw" is 0, the program adds data.
                 break;
        case 1 : ans = a - b;       <- If the content of "sw" is 1, the program subtracts data.
                 break;
        case 2 : ans = a * b;       <- If the content of "sw" is 2, the program multiplies data.
                 break;
        case 3 : ans = a / b;       <- If the content of "sw" is 3, the program divides data.
                 break;
        default: error();           <- If the content of "sw" is 4 or greater,
                 break;                the program performs error processing.
    }
}
```

Column switch-case Statement without Break

A switch-case statement normally has a break statement entered at the end of each of its execution statements.
If a block that is not accompanied by a break statement is encountered, the program executes the next block after terminating that block. In this way, blocks are executed sequentially from above. Therefore, this allows the start position of processing to be changed depending on the value of an expression.



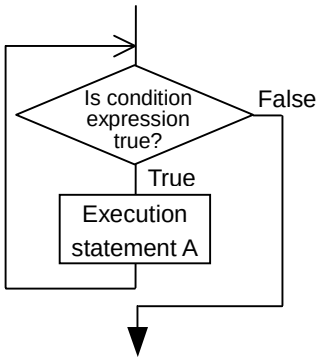

```
switch(expression) {

    case constant 1:    Execution statement A

    case constant 2:    Execution statement B

    case constant 3:    Execution statement C

    default:            Execution statement D

}
```

1.4.3 Repetition of Same Processing (Repeat Processing)

Control statements used to write repeat processing include "while", "for" and "do-while" statements. This section explains how to write these control statements and shows some examples of usage.

while Statement

This statement executes processing in a block repeatedly as long as the given condition expression is met. An endless loop can be implemented by writing a constant other than 0 in the condition expression, because the condition expression in this case is always "true".




```
while(condition expression) {

    Execution statement A

}
```

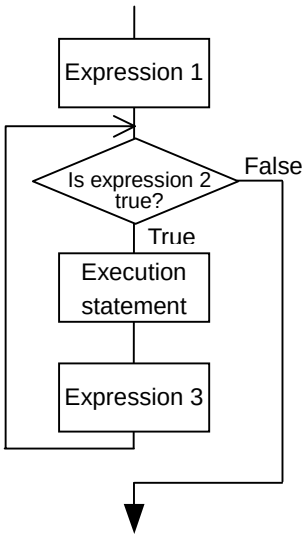
Finding Sum Total –1– (while Statement)

In this example, the program finds the sum of integers from 1 to 100.

```
void sum(void);                  <- Declares "sum" function.
                                    (Refer to Section 1.5, "Functions".)
unsigned int total = 0;          <- Declares the variables used.

void sum(void)                   <- Defines "sum" function.
{
    unsigned int i = 1;          <- Defines and initializes counter variables.

    while(i <= 100){             <- Loops until the counter content reaches 100.
        total += i;
        i++;                     <- Changes the counter content.
    }
}
```

for Statement

The repeat processing that is performed by using a counter always requires operations to "initialize" and "change" the counter content, in addition to determining the given condition. A for statement makes it possible to write these operations along with a condition expression. Initialization (expression 1), condition expression (expression 2), and processing (expression 3) each can be omitted. However, when any of these expressions is omitted, make sure the semicolons (;) placed between expressions are left in. This for statement and the while statement described above can always be rewritten.




```
for (expression 1; expression 2; expression 3) {

        Execution statement

}
```

Finding Sum Total –2– (for Statement)

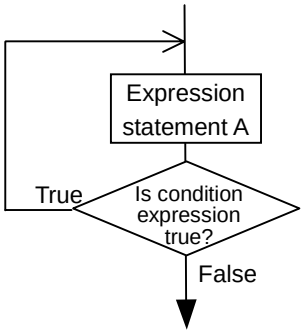
In this example, the program finds the sum of integers from 1 to 100.

```
void sum(void);
unsigned int total = 0;
void sum(void)
{
    unsigned int i = 1;
    for(i = 1; i <= 100; i++){
        total += i;
    }
}
```

Declares "sum" function.
(Refer to Section 1.5, "Functions".)
Declares the variables used.
Defines "sum" function.
Defines counter variables.
Loops until the counter content increments from 1 to 100.

do-while Statement

Unlike the for and while statements, this statement determines whether a condition is true or false after executing processing (post-execution determination). Although there could be some processing in the for or while statements that is never once executed, all processing in a do-while statement is executed at least once.




```
do{
        expression statement
} while(condition expression);
```

Finding Sum Total –3– (do-while Statement)

In this example, the program finds the sum of integers from 1 to 100.

```
void sum(void);
unsigned int total = 0;
void sum(void)
{
    unsigned int i = 1;
    do{
        i ++;
        total += i;
    }while(i < 100);
}
```

Declares "sum" function.
(Refer to Section 1.5, "Functions".)

Declares the variables used.

Defines "sum" function.

Defines and initializes counter variables.

Loops until the counter content increments from 1 to 100.

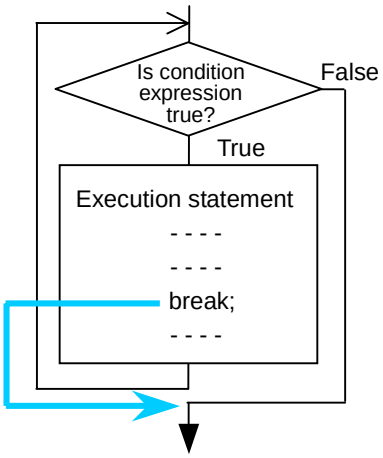
1.4.4 Suspending Processing

There are control statements (auxiliary control statements) such as break, continue, and goto statements that make it possible to suspend processing and quit.
This section explains how to write these control statements and shows some examples of usage.

break Statement

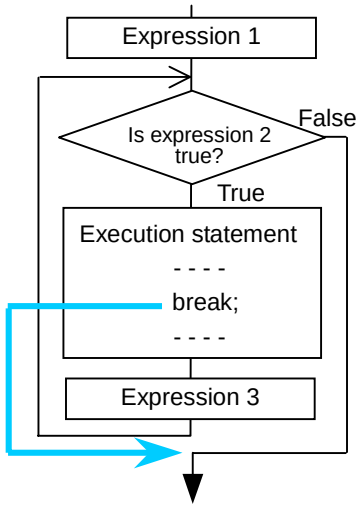
Use this statement in repeat processing or in a switch-case statement. When "break;" is executed, the program suspends processing and exits only one block.

• When used in a while statement




```
while (condition expression) {
                - - - - -
                - - - - -
                break;
                - - - - -
}
for (expression 1;
expression 2; expression 3) {
                - - - - -
                break;
                - - - - -
}
```

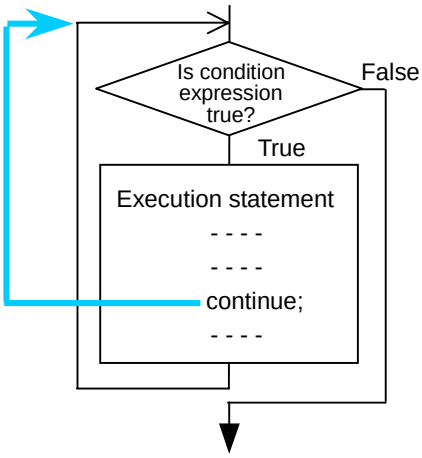
• When used in a for statement



continue Statement

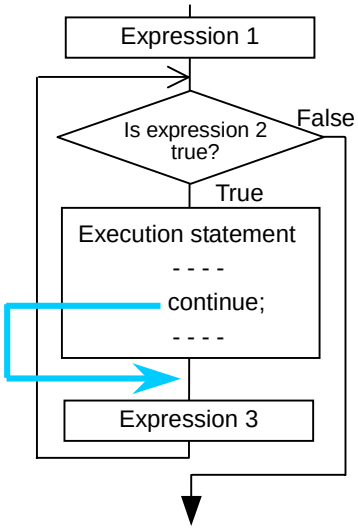
Use this statement in repeat processing. When "continue;" is executed, the program suspends processing. After being suspended, the program returns to condition determination when continue is used in a while statement or executes expression 3 before returning to condition determination when used in a for statement.

• When used in a while statement



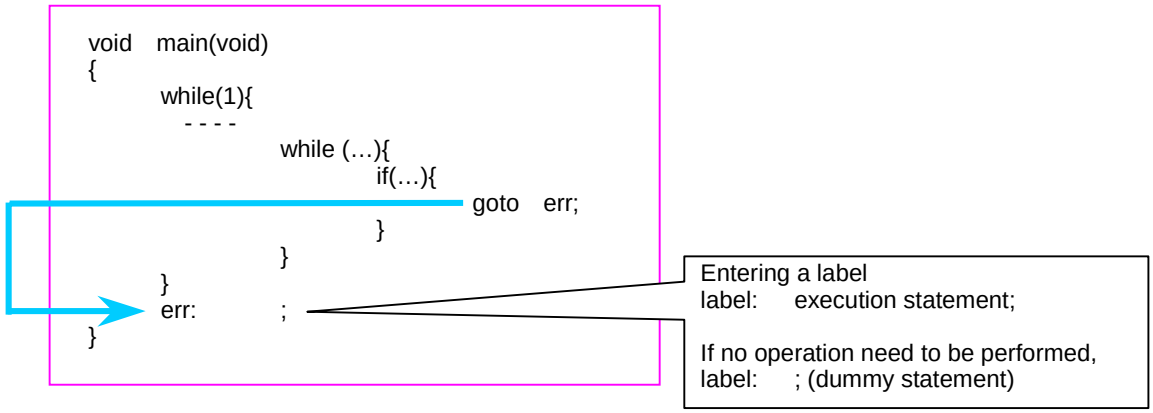

```
while (condition expression) {
                - - - - -
                - - - - -
                continue;
                - - - - -
}
for (expression 1;
expression 2; expression 3) {
                - - - - -
                continue;
                - - - - -
}
```

• When used in a for statement



goto Statement

When a goto statement is executed, the program unconditionally branches to the label written after the goto statement. Unlike break and continue statements, this statement makes it possible to exit multiple blocks collectively and branch to any desired location in the function. However, since this operation is contrary to structured programming, it is recommended that a goto statement be used in only exceptional cases as in error processing. Note also that the label indicating a jump address must always be followed by an execution statement. If no operation need to be performed, write a dummy statement (only a semicolon ';') after the label.



1.5 Functions

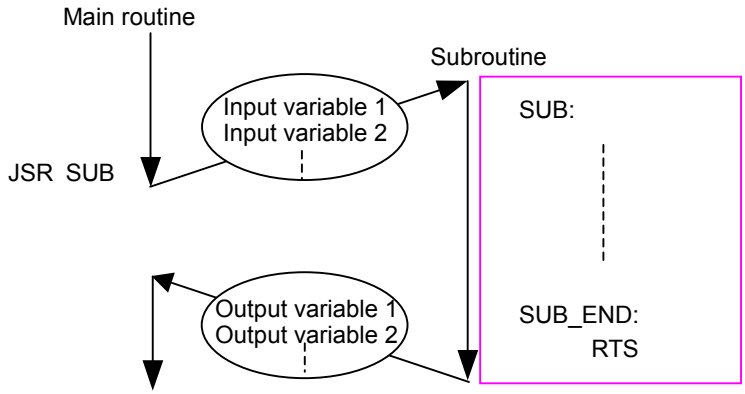
1.5.1 Functions and Subroutines

As subroutines are the basic units of program in the assembly language, so are the "functions" in the C language.
This section explains how to write functions in ICC740.

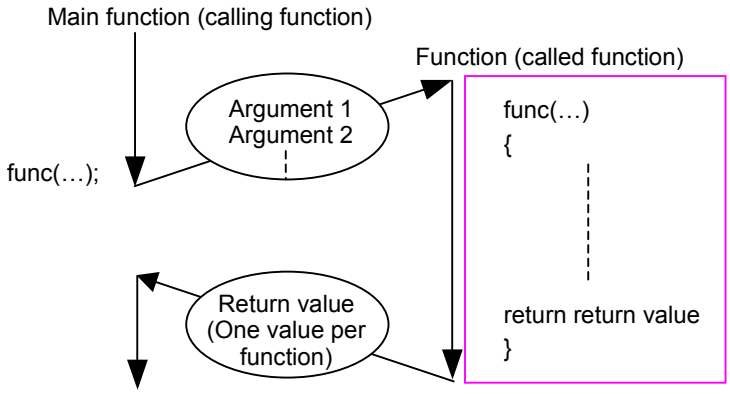
Arguments and Return Values

Data exchanges between functions are accomplished by using "arguments", equivalent to input variables in a subroutine, and "return values", equivalent to output variables in a subroutine.
In the assembly language, no restrictions are imposed on the number of input or output variables. In the C language, however, there is a rule that one return value per function is accepted, and a "return statement" is used to return the value.
About the argument, the size of the argument is decided up to a total of 256 bytes per one function, and when the size is over 256 bytes, the compiler generates an error.

• "Subroutine" in assembly language



• "Function" in C language



1.5.2 Creating Functions

Three procedures are required before a function can be used. These are "function declaration" (prototype declaration), "function definition", and "function call".
This section explains how to write these procedures.

Function Declaration (Prototype Declaration)

Before a function can be used in the C language, function declaration (prototype declaration) must be entered first. The type of function refers to the data types of the arguments and the returned value of a function.
The following shows the format of function declaration (prototype declaration):

```
data type of returned value      function name (list of data types of arguments);
```

If there is no returned value and argument, write the type called "void" that means null.

Function Definition

In the function proper, define the data types and the names of "dummy arguments" that are required for receiving arguments. Use the "return statement" to return the value for the argument.
The following shows the format of function definition:

```
data type of return value     function name (data type of dummy argument 1 dummy argument 1, ....)
{
                ¦
        return   return   value;
}
```

Function Call

When calling a function, write the argument for that function. Use a assignment operator to receive a return value from the called function.

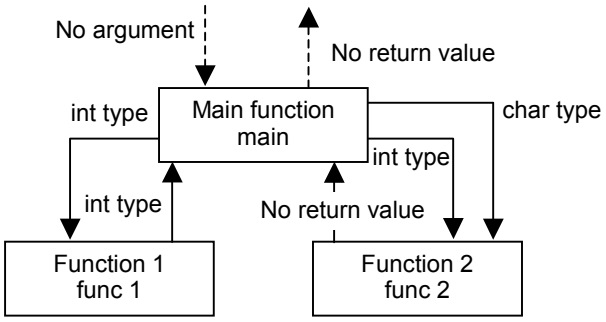
```
function name (argument 1, ...);
```

When there is a return value

```
variable = function name (argument 1, ...);
```

Example for a Function

In this example, we will write three functions that are interrelated as shown below.




```
/*   Prototype declaration   */
void   main(void);
int   func1(int);
void   func2(int, char);

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

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

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

Calls function 1 ("func1") using a as argument.
Return value is substituted for "ans".

Calls function 2 ("func2") using b, c as arguments.
There is no return value.

Returns a value for the argument
using a "return statement".

1.5.3 Exchanging Data between Functions

In the C language, exchanges of arguments and return values between functions are accomplished by copying the value of each variable as it is passed to the receiver ("Call by Value"). Consequently, the name of the argument used when calling a function and the name of the argument (dummy argument) received by the called function do not need to coincide.
Since processing in the called function is performed using copied dummy arguments, there is no possibility of damaging the argument proper in the calling function.
For these reasons, functions in the C language are independent of each other, making it possible to reuse the functions easily.
This section explains how data are exchanged between functions.

Finding Sum of Integers (Example for a Function)

In this example, using two arbitrary integers in the range of -32,768 to 32,767 as arguments, we will create a function "add" to find a sum of those integers and call it from the main function.

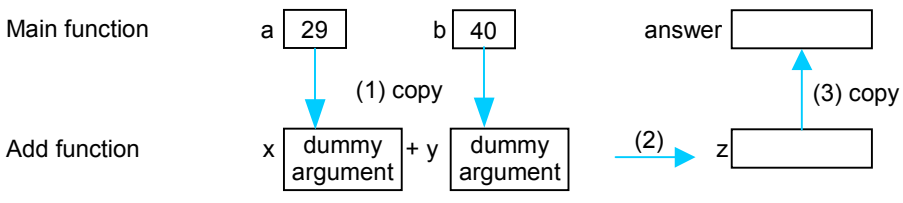
```
/*   Prototype declaration   */
void   main(void);
long   add(int, int);

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

      answer = add(a, b);
}

/*   Add function   */
long   add(int   x, int   y)
{
      long int   z;

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

(1) Calls the add function.

(2) Executes addition.

(3) Returns a value for the argument.

<Flow of data>



1.6 Storage Classes

1.6.1 Effective Range of Variables and Functions

Variables and functions have different effective ranges depending on their nature, e.g., whether they are used in the entire program or in only one function. These effective ranges of variables and functions are called "storage classes (or scope)".
This section explains the types of storage classes of variables and functions and how to specify them.

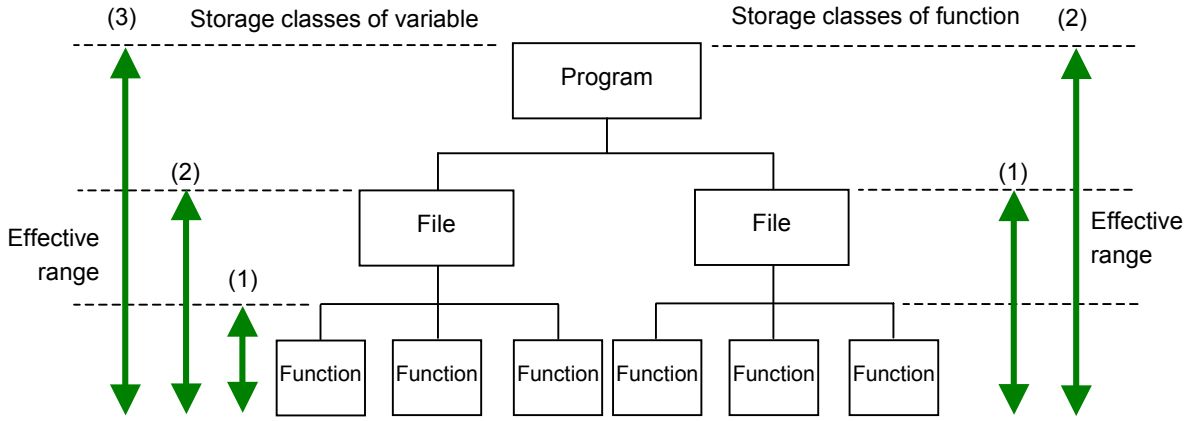
Effective Range of Variables and Functions

A C language program consists of multiple source files. Furthermore, each of these source files consists of multiple functions. Therefore, a C language program is hierarchically structured as shown below.

There are following three storage classes for a variable:
(1) Effective in only a function
(2) Effective in only a file
(3) Effective in the entire program

There are following two storage classes for a function:
(1) Effective in only a file
(2) Effective in the entire program

In the C language, these storage classes can be specified for each variable and each function. Effective utilization of these storage classes makes it possible to protect the variables or functions that have been created or conversely share them among the members of a team.

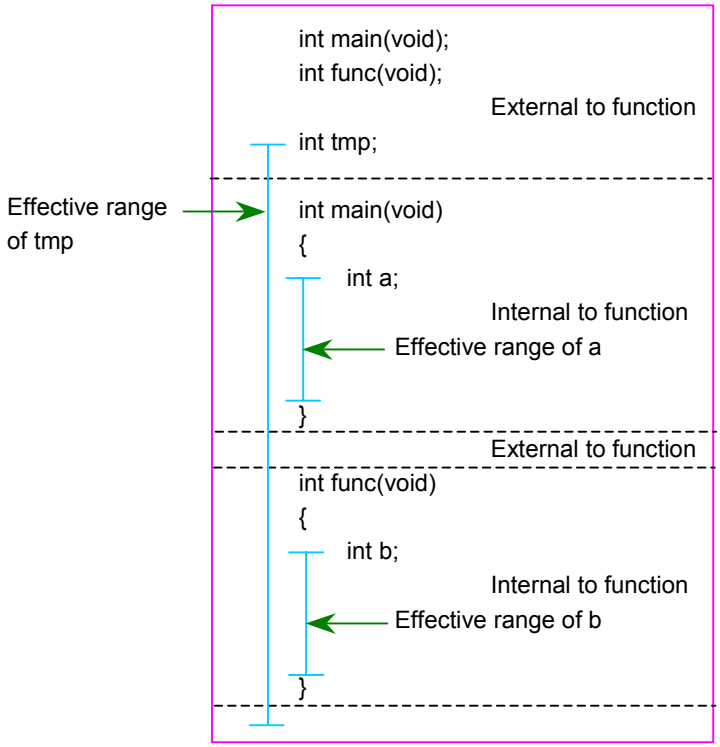


1.6.2 Storage Classes of Variables

The storage class of a variable is specified when writing type declaration. There are following two points in this:
(1) External and internal variables (→location where type declaration is entered)
(2) Storage class specifier (→specifier is added to type declaration)
This section explains how to specify storage classes for variables.

External and Internal Variables

This is the simplest method to specify the effective range of a variable. The variable effective range is determined by a location where its type declaration is entered. Variables declared outside a function are called "external variables" and those declared inside a function are called "internal variables". External variables are global variables that can be referenced from any function following the declaration. Conversely, internal variables are local variables that can be effective in only the function where they are declared following the declaration.



Storage Class Specifiers

The storage class specifiers that can be used for variables are auto, static, register, and extern. These storage class specifiers function differently when they are used for external variables or internal variables. The following shows the format of a storage class specifier.

storage class specifier Δ data type Δ variable name;

Storage Classes of External Variable

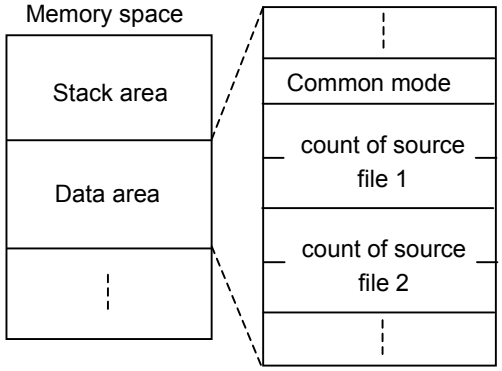
If no storage class specifier is added for an external variable when declaring it, the variable is assumed to be a global variable that is effective in the entire program. On the other hand, if an external variable is specified of its storage class by writing "static" when declaring it, the variable is assumed to be a local variable that is effective in only the file where it is declared. Write the specifier "extern" when using an external variable that is defined in another file like "mode" in source file 2 of the following.
External variables which do not set the initial value are assigned to the N_UDATA segment on data area. External variables which set the initial value are assigned to the N_IDATA segment on data area.

Source file 1

```
char mode;
static int count;

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

Source file 2

```
extern char mode;
static int count;

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

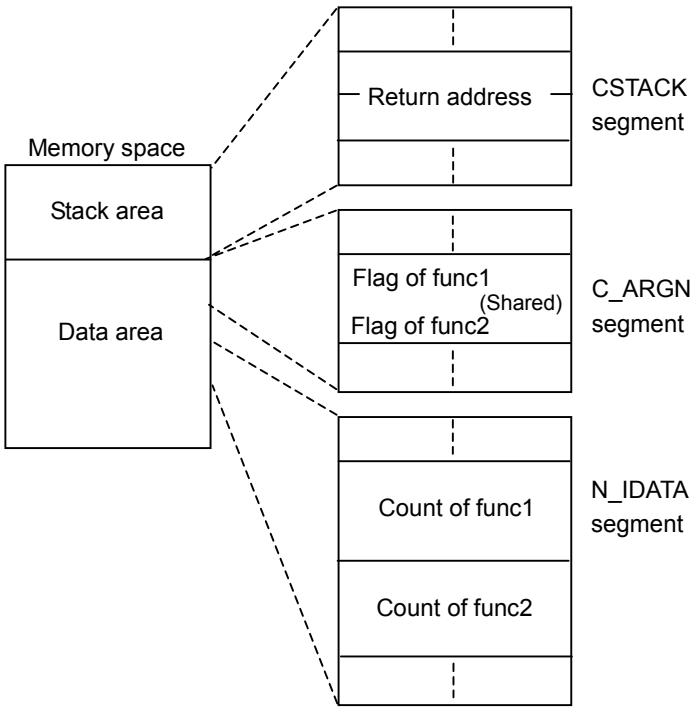


Storage Classes of Internal Variable

An internal variable declared without adding any storage class specifier has its area allocated in C-ARGN segment on the data area. Therefore, such a variable is shared with each function ad it is initialized each time the function is called.
On the other hand, internal variables whose storage class is specified to be "static", which do not set the initial value are assigned to the N_UDATA segment on data area and which set the initial value are assigned to the N_IDATA segment on data area. The variable is initialized only once when starting up the program.

Source file

```
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 Storage Classes of Functions

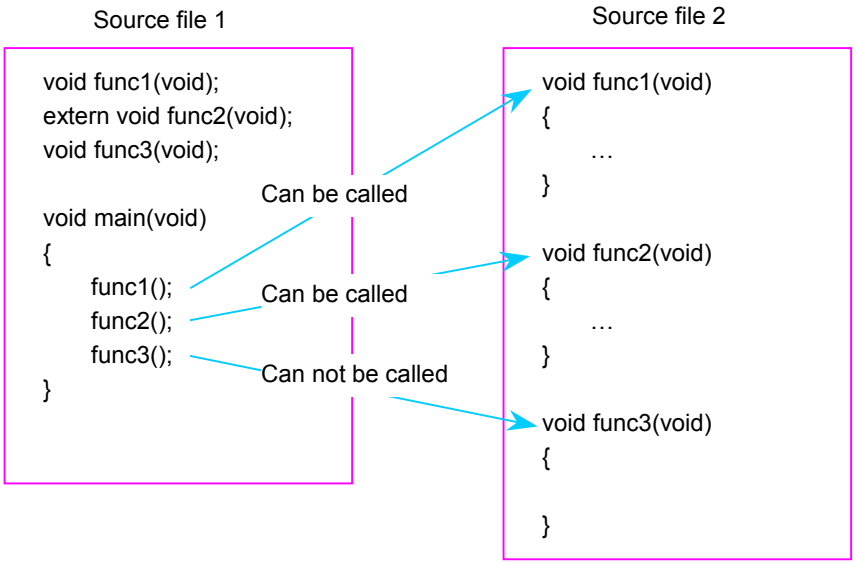
The storage class of a function is specified on both function defining and function calling sides. The storage class specifiers that can be used here are static and extern. This section explains how to specify the storage class of a function.

Global and Local Functions

(1) If no storage class is specified for a function when defining it
This function is assumed to be a global function that can be called and used from any other source file.

(2) If a function is declared to be "static" when defining it
This function is assumed to be a local function that cannot be called from any other source file.

(3) If a function is declared to be "extern" in its type declaration
This storage class specifier indicates that the declared function is not included in the source file where functions are declared, and that the function in some other source file be called. However, only if a function has its type declared--even though it may not be specified to be "extern", if the function is not found in the source file, the function in some other source file is automatically called in the same way as when explicitly specified to be "extern".



Summary of Storage Classes

Storage classes of variables and storage classes of functions are summarized below.

Storage Classes of Variables

Storage class	External variable	Internal variable
Storage class specifiers omitted	Global variables that can also be referenced from other source files. [Allocated in segment N_UDATA and N_IDATA] [Maintain data]	Variables that are effective in only the function. [Allocated in a segment C_ARGN when executing the function.] [Not maintain data]
auto		Variables that are effective in only the function. [Allocated in a segment C_ARGN when executing the function.] [Not maintain data]
static	Local variables that cannot be referenced from other source files. [Allocated in segment N_UDATA and N_IDATA] [Maintain data]	Variables that are effective in only the function. [Allocated in segment N_UDATA and N_IDATA.] [Maintain data]
register		Variables that are effective in only the function. [Allocated in segment C_ARGN.] [Not maintain data]
extern	Variables that reference variables in other source files. [Not allocated in memory] [Maintain data]	Variables that reference variables in other source files. (cannot be referenced from other functions.) [Not allocated in the memory]

Storage Classes of Functions

Storage class	Types of functions
Storage class specifiers omitted	Global functions that can be called and executed from other source files [Specified on function defining side]
static	Local functions that can not be called and executed from other source files [Specified on function defining side]
extern	Calls a function in other source files [Specified on function calling side]

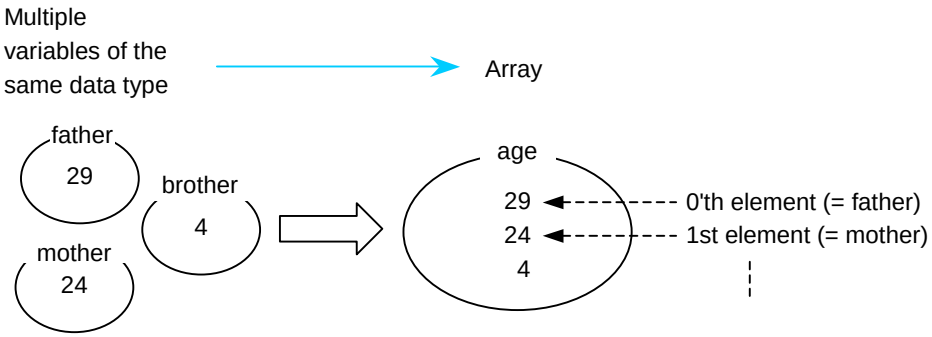
1.7 Arrays and Pointers

1.7.1 Arrays

Arrays and pointers are the characteristic features of the C language.
This section describes how to use arrays and explains pointers that provide an important means of handling the array.

What is an Array?

The following explains the functionality of an array by using a program to find the total age of family members as an example. The family consists of parents (father = 29 years old, mother = 24 years old), and a child (brother = 4 years old).
In this program, the number of variable names increases as the family grows. To cope with this problem, the C language uses a concept called an "array". An array is such that data of the same type (int type) are handled as one set. In this example, father's age (father), mother's age (mother), and child's age (brother) all are not handled as separate variables, but are handled as an aggregate as family age (age). Each data constitutes an "element" of the aggregate. Namely, the 0'th element is father, the 1st element is mother, and the 2nd element is the brother.



Example Finding Total Age of a family (1)

In this example, we will find the total age of family members (father, mother and brother).

As the family grows, so do the type declaration of variables and the execution statements to be initialized.

```
void main(void)
{
        int father = 29;
        int mother = 24;
        int brother = 4;
        int total;

        total = father + mother + brother;
}
```

```
void main(void)
{
        int father = 29;
        int mother = 24;
        int brother = 4;
        int sister 1 = 1;
        int sister 2 = 1;
              :
        int total;

        total = father + mother + brother + sister1 + sister2 + …;
}
```

1.7.2 Creating an Array

There are two types of arrays handled in the C language: "one-dimensional array" and "two-dimensional array".
This section describes how to create and reference each type of array.

One-dimensional Array

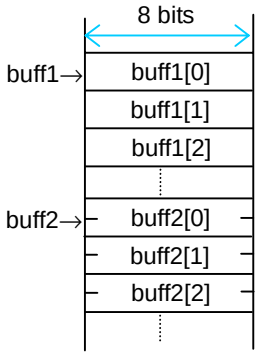
A one-dimensional array has a one-dimensional (linear) expanse. The following shows the declaration format of a one-dimensional array.

```
Data type    array name [number of elements];
```

When the above declaration is made, an area is allocated in memory for the number of elements, with the array name used as the beginning label.
To reference a one-dimensional array, add element numbers to the array name as subscript. However, since element numbers begin with 0, the last element number is 1 less than the number of elements.

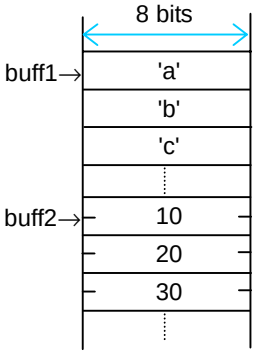
• Declaration of one-dimensional array

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



• Declaration and initialization of one-dimensional array

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

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



Finding Total Age of a Family (2)

In this example, we will find the total age of family members by using an array.

```
#define MAX 3 (Note)

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

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

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

or

```
#define MAX 3

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

        int total = 0;
        int i;

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

Initialized simultaneously when declared.

By using an array, it is possible to utilize a repeat statement where the number of elements are used as variables.

(Note): #define MAX 3: Synonym defined as MAX = 3. (Refer to Section 1.9, Preprocess Commands".)

Two-dimensional Array

A two-dimensional array has a planar expanse comprised of "columns" and "rows". Or it can be considered to be an array of one-dimensional arrays. The following shows the declaration format of a two-dimensional array.

```
Data type   array name [number of rows] [number of columns];
```

To reference a two-dimensional array, add "row numbers" and "column numbers" to the array name as subscript. Since both row and column numbers begin with 0, the last row (or column) number is 1 less than the number of rows (or columns).

• Concept of two-dimensional array

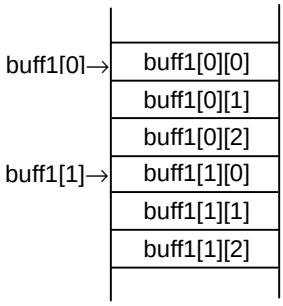
Columns→

Rows ↓

Row 0 Column 0	Row 0 Column 1	Row 0 Column 2	Row 0 Column 3
Row 1 Column 0	Row 1 Column 1	Row 1 Column 2	Row 1 Column 3
Row 2 Column 0	Row 2 Column 1	Row 2 Column 2	Row 3 Column 3

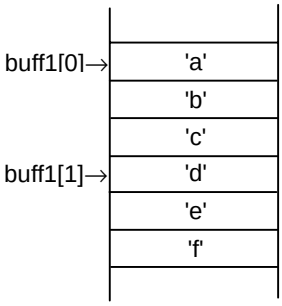
• Declaration and initialization of two-dimensional array

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

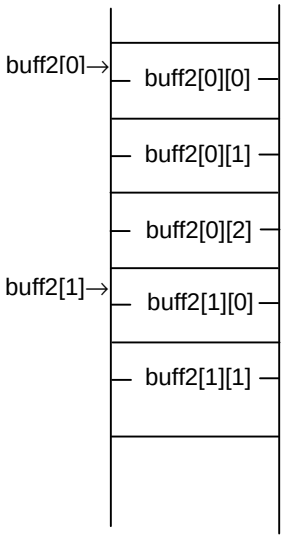


• Declaration and initialization of two-dimensional array

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

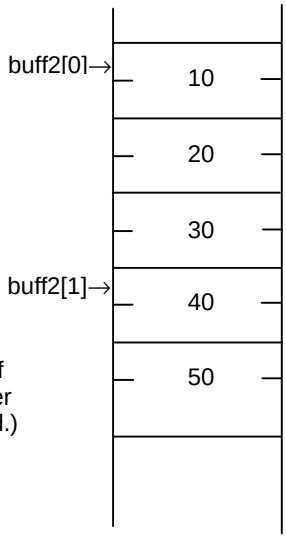



```
int buff2[2][3];
```




```
int buff2[2][3] = {
   10, 20, 30, 40, 50, 60
};
```

When initializing a two-dimensional array simultaneously with declaration, specification of the number of rows can be omitted. (Number of columns cannot be omitted.)



1.7.3 Pointers

A pointer is a variable that points to data; i.e., it indicates an address.
A "pointer variable" which will be described here handles the "address" at which data is stored as a variable. This is equivalent to what is referred to as "indirect addressing" in assembly language.
This section explains how to declare and reference a pointer variable.

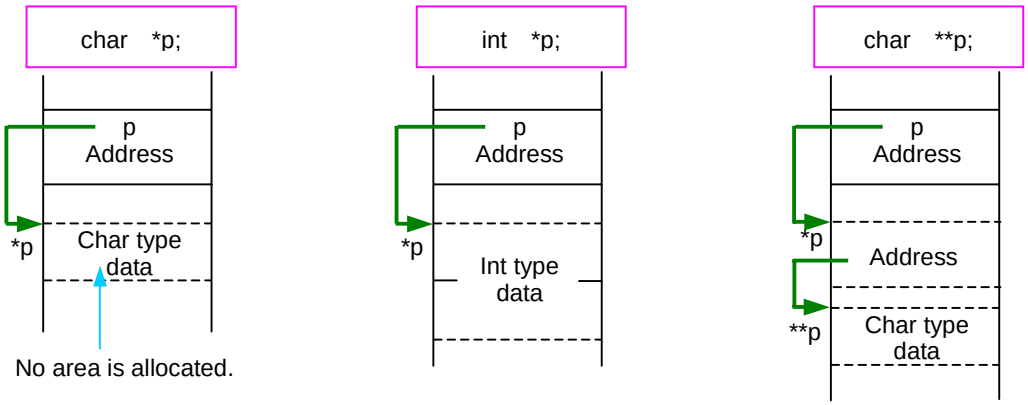
Declaring a Pointer Variable

The format show below is used to declare a pointer variable.

Pointed data type ＊pointer variable name;

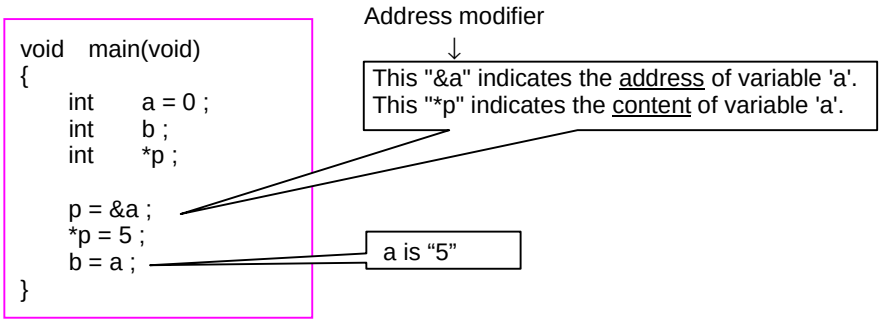
However, it is only an area to store an address that is allocated in memory by the above declaration. For the data proper to be assigned an area, it is necessary to write type declaration separately.

• Pointer variable declaration



Relationship between Pointer Variables and Variables

The following explains the relationship between pointer variables and variables by using a method for substituting constant '5' by using pointer variable 'p' for variable of int type 'a' as an example.

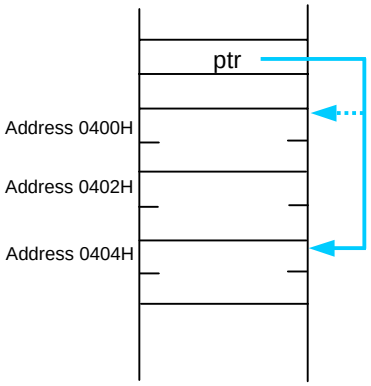


Operating on Pointer Variables

Pointer variables can be operated on by addition or subtraction. However, operation on pointer variables differs from operation on integers in that the result is an address value. Therefore, address values vary with the data size indicated by the pointer variable.

Address + (integer X sizeof (type))
Address - (integers X izeof (type))

```
int   * ptr;

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



The pointer variable ptr is an int type of variable. When calculated by sizeof(int), the size of the int-type variable is found to be 2 bytes. Therefore, p + 2 points to address 0404H.

Column Data Length of Pointer Variable

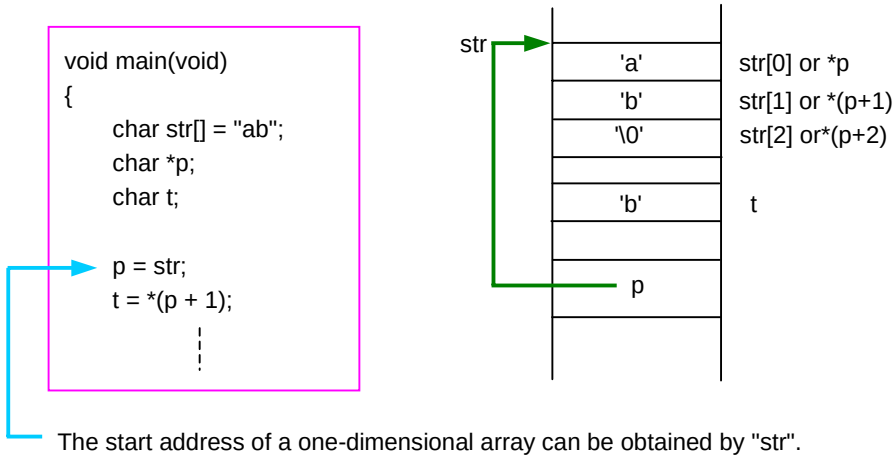
The data length of variables in C language programs are determined by the data type. For a pointer variable, since its content is an address, the data length provided for it is sufficiently large to represent the entire address space that can be accessed by the microprocessor used.

1.7.4 Using Pointers

This section shows some examples for effectively using a pointer.

Pointer Variables and One-dimensional Array

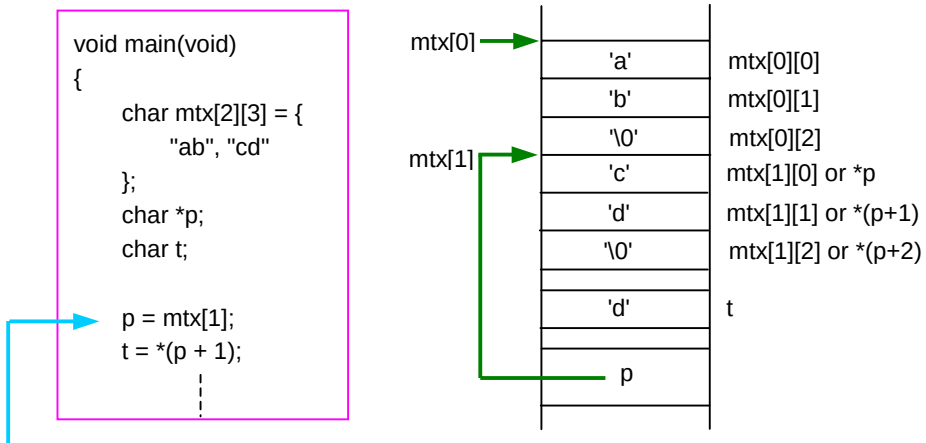
When an array is declared by using subscripts to indicate its element numbers, it is encoded as "index addressing". In this case, therefore, address calculations to determine each address "as reckoned from the start address" are required whenever accessing the array.
On the other hand, if an array is declared by using pointer variables, it can be accessed in indirect addressing.



The start address of a one-dimensional array can be obtained by "str".
(Address modifier '&' is unnecessary.)

Pointer Variables and Two-dimensional Array

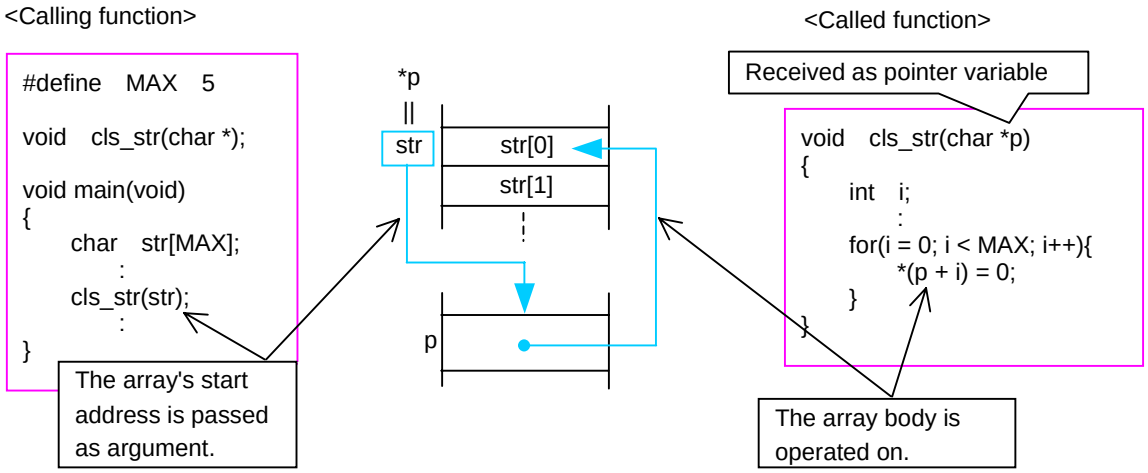
As in the case of a one-dimensional array, a two-dimensional array can also be accessed by using pointer variables.



The start address of the first row of a two-dimensional array "mtx" can be obtained by "mtx[1]". ('&' is unnecessary.)

Passing Addresses between Functions

The basic method of passing data to and from C language functions is referred to as "Call by Value". With this method, however, arrays and character strings cannot be passed between functions as arguments or returned values.
Used to solve this problem is a method, known as "Call by Reference", which uses a pointer variable. In addition to passing the addresses of arrays or character strings between functions, this method can be used when it is necessary to pass multiple data as a returned value.
Unlike the Call by Value method, this method has a drawback in that the independency of each function is reduced, because the data in the calling function is rewritten directly.
The following shows an example where an array is passed between functions using the Call by Reference method.



Column Passing Data between Functions at High

In addition to the Call by Value and the Call by Reference methods, there is another method to pass data to and from functions. With this method, the data to be passed is turned into an external variable.
This method results in loosing the independency of functions and, hence, is not recommended for use in C language programs. Yet, it has the advantage that functions can be called at high speed because entry and exit processing (argument and return value transfers) normally required when calling a function are unnecessary. Therefore, this method is frequently used in ROM'ed programs where general-purpose capability is not an important requirement and the primary concern is high-speed processing.

1.7.5 Placing Pointers into an Array

This section explains a "pointer array" where pointer variables are arranged in an array.

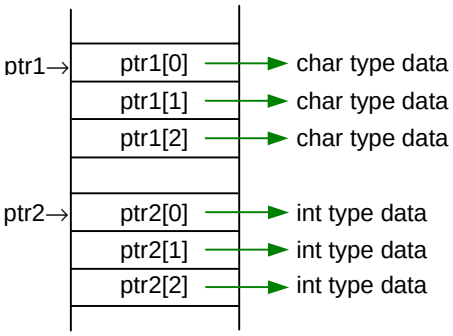
Pointer Array Declaration

The following shows how to declare a pointer array.

Data type ∗array name [number of elements];

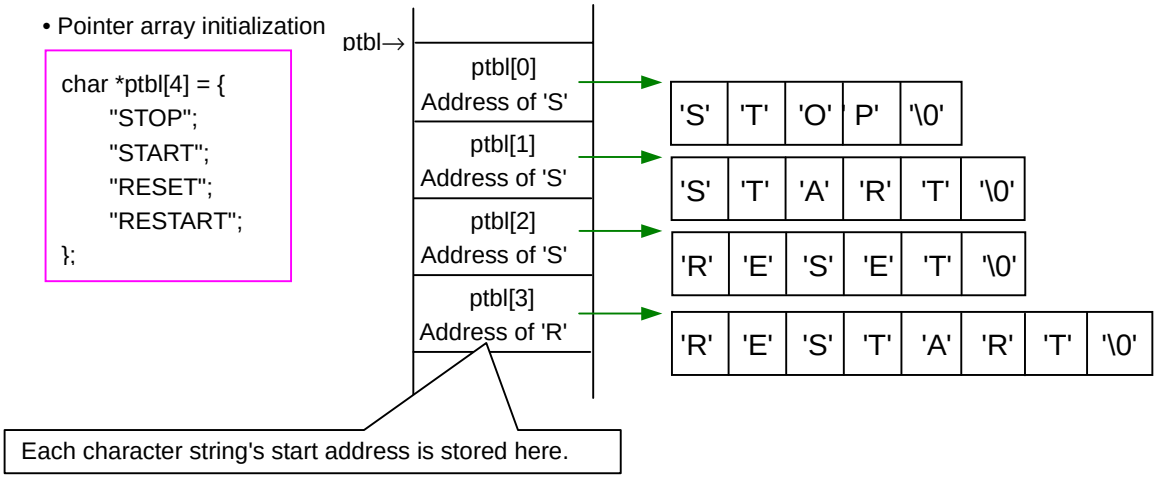
• Pointer array declaration

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



• Pointer array initialization

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



Pointer Array and Two-dimensional Array

The following explains the difference between a pointer array and a two-dimensional array. When multiple character strings each consisting of a different number of characters are declared in a two-dimensional array, the free spaces are filled with null code "\0". If the same is declared in a pointer array, there is no free space in memory. For this reason, a pointer array is a more effective method than the other type of array when a large amount of character strings need to be operated on or it is necessary to reduce memory requirements to a possible minimum.

• Two-dimensional array

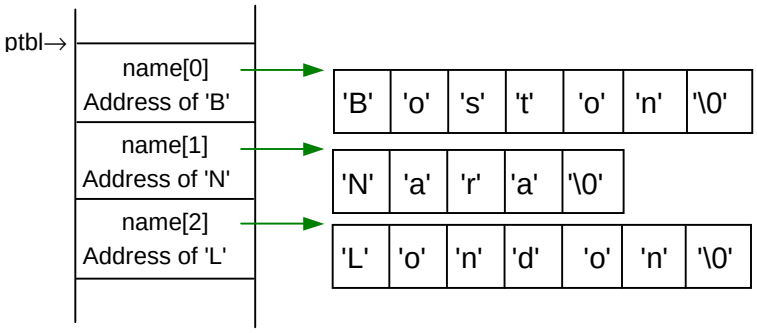
```
char name[3][7] = {
        "Boston",
        "Nara",
        "London"
};
```

'B'	'o'	's'	't'	'o'	'n'	'\0'
'N'	'a'	'r'	'a'	'\0'	'\0'	'\0'
'L'	'o'	'n'	'd'	'o'	'n'	'\0'

Filled with null code.

• Pointer array

```
char    *name[3] = {
        "Boston",
        "Nara",
        "London"
};
```



1.7.6 Table Jump Using Function Pointer

In assembly language programs, "table jump" is used when switching processing load increases depending on the contents of some data. The same effect as this can be obtained in C language programs also by using the pointer array described above.
This section explains how to write a table jump using a "function pointer".

What Does a Function Pointer Mean?

A "function pointer" is one that points to the start address of a function in the same way as the pointer described above. When this pointer is used, a called function can be turned into a parameter. The following shows the declaration and reference formats for this pointer.

<Declaration format> Type of return value (*function pointer name) (data type of argument);
<Reference format> Variable in which to store return value = (*function pointer name) (argument);

Switching Arithmetic Operations Using Table Jump

The method of calculation is switched over depending on the content of variable "num".

```
/* Prototype declaration ***************/
int   calc_f(int, int, int);
int   add_f(int, int), sub_f(int, int);
int   mul_f(int, int), div_f(int, 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;
}
```

Function pointers arranged in an array

jmptbl[0]	Start address of "add_f
jmptbl[1]	Start address of "sub_f
jmptbl[2]	Start address of "mul_f
jmptbl[3]	Start address of "div_f

Setting of jump address

Function call using a function pointer

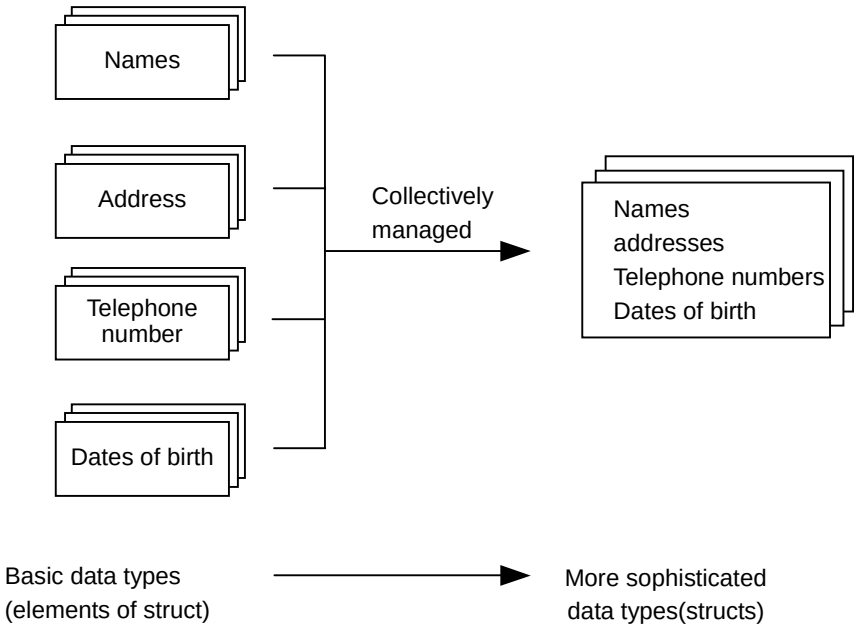
1.8 Struct and Union

1.8.1 Struct and Union

The data types discussed hereto (e.g., char, signed int, and unsigned int types) are called the "basic data types" stipulated in compiler specifications.
The C language allows the user to create new data types based on these basic data types. These are "struct" and "union".
The following explains how to declare and reference structs and unions.

From Basic Data Types to Structs

Structs and unions allows the user to create more sophisticated data types based on the basic data types according to the purposes of use. Furthermore, the newly created data types can be referenced and arranged in an array in the same way as the basic data types.



1.8.2 Creating New Data Types

The elements that constitute a new data type are called "members". To create a new data type, define the members that constitute it. This definition makes it possible to declare a data type to allocate a memory area and reference it as necessary in the same way as the variables described earlier.
This section describes how to define and reference structs and unions, respectively.

Difference between Struct and Union

When allocating a memory area, members are located differently for structs and unions.
(1) Struct: Members are sequentially located.
(2) Union: Members are located in the same address.
(Multiple members share the same memory area. The union size is the largest size in the members which are assigned to the same address.)

Definition and Declaration of Struct

To define a struct, write "struct".

```
struct   struct tag   {
         Member 1;
         Member 2;
              :
};
```

The above description creates a data type "struct struct tag". Declaration of a struct with this data type allocates a memory area for it in the same way as for an ordinary variable.

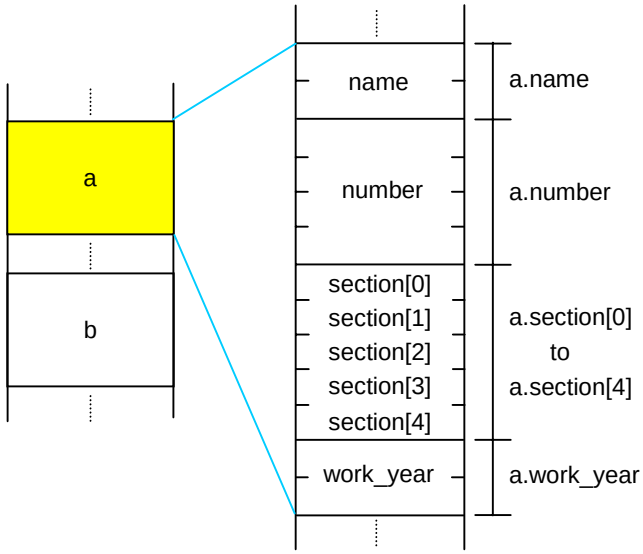
```
struct   struct tag   struct variable name;
```

Referencing Struct

To refer to each member of a struct, use a period '.' that is a struct member operator.

struct variable name.member name

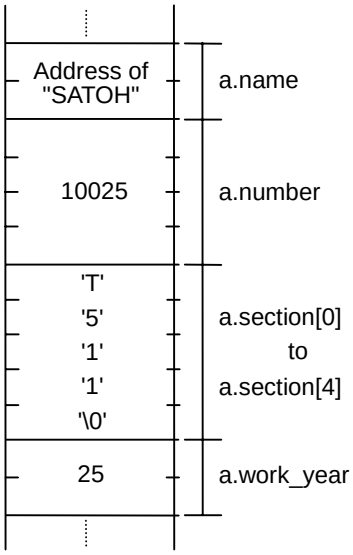
```
struct   person{
      char   *name;
      long   number;
      char   section[5];
      int    work_year;
};

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



The initial data of each member is written and arranged according to the declaration order (following types) when structure variables are initialized.

- Initialization of struct variable

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

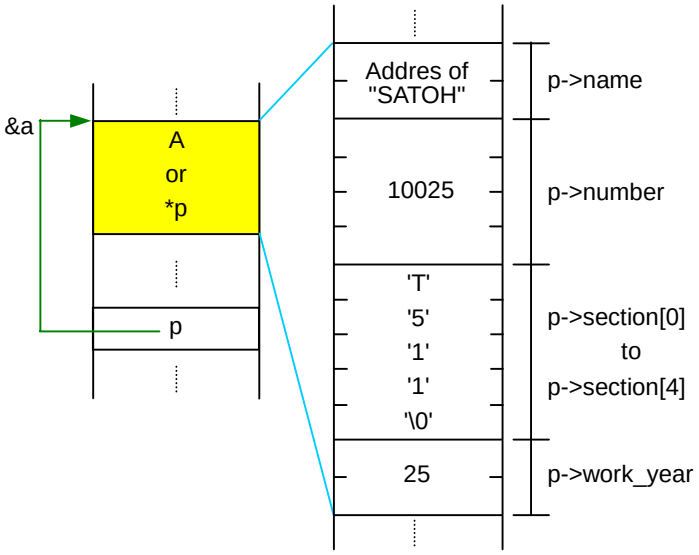


Example for Referencing Members Using a Pointer

To refer to each member of a struct using a pointer, use an arrow '->'.

Pointer->member name

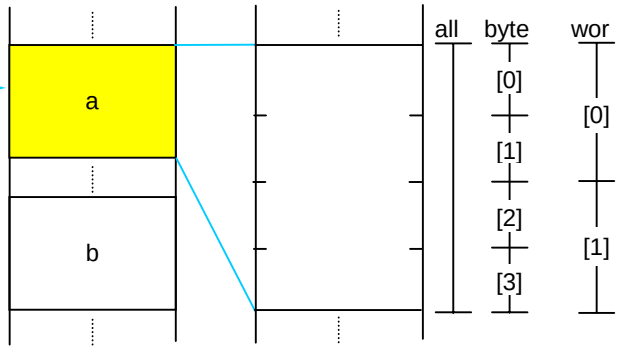
```
#define   LYEAR   20
struct   person{
      char   *name;
      long   number;
      char   section[5];
      int   work_year;
};

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

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



Unions

Unions are characteristic in that an allocated memory area is shared by all members. Therefore, it is possible to save on memory usage by using unions for multiple entries of such data that will never exist simultaneously. Unions also will prove convenient when they are used for data that needs to be handled in different units of data size, e.g., 16 bits or 8 units, depending on situation.
To define a union, write "union". Except this description, the procedures for defining, declaring, and referencing unions all are the same as explained for structs.

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

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



A 4-byte area is shared by all, byte, and word.

Column Type Definition

Since structs and unions require the keywords "struct" and "union", there is a tendency that the number of characters in defined data types increases. One method to circumvent this is to use a type definition "typedef".

```
typedef   existing type name   new type name;
```

When the above description is made, the new type name is assumed to be synonymous with the existing type name and, therefore, either type name can be used in the program.
The following shows an example of how "typedef" can actually be used.

When defining types, structure (union) tag names can be omitted.

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

struct data   sdata, *sptr;
        :
```

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

DATA   sdata, *sptr;
        :
```

1.9 Preprocess Commands

1.9.1 Preprocess Commands of ICC740

The C language supports file inclusion, macro function, conditional compile, and some other functions as "preprocess commands".
The following explains the main preprocess commands available with ICC740.

Preprocess Command List of ICC740

Preprocess commands each consist of a character string that begins with the symbol '#' to discriminate them from other execution statements. Although they can be written at any position, the semicolon ';' to separate entries is unnecessary. The following lists the main preprocess commands that can be used in ICC740.

Description	Function
#include	Takes in a specified file.
#define	Replaces character string and defines macro.
#undef	Cancels definition made by #define.
#if to #elif to #else to #endif	Performs conditional compile.
#ifdef to #elif to #else to #endif	Performs conditional compile.
#ifndef to #elif to #else to #endif	Performs conditional compile.
#error	Outputs message to standard output devices before suspending processing.
#line	Specifies a file's line numbers.
#pragma	Instructs processing of ICC740's extended function.

1.9.2 Including a File

Use the command "#include" to take in another file. ICC740 requires different methods of description depending on the directory to be searched.
This section explains how to write the command "#include" for each purpose of use.

Searching for Standard Directory

```
#include    <file name>
```

This statement takes in a file from the directory specified with the startup option '–I.' If the specified file does not exist in this directory, ICC740 searches the standard directory that is set with ICC740's environment variable "C_INCLUDE" as it takes in the file.
As the standard directory, normally specify a directory that contains the "standard include file".

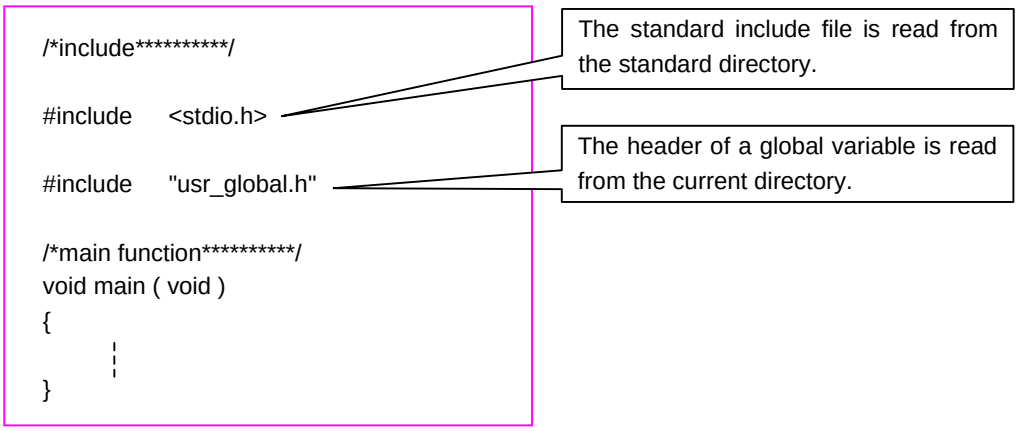
Searching for Current Directory

```
#include    "file name"
```

This statement takes in a file from the current directory. If the specified file does not exist in the current directory, ICC740 searches the directory specified with the startup option '–I' and the directory set with ICC740's environment variable " C_INCLUDE" in that order as it takes in the file.
To discriminate your original include file from the standard include file, place that file in the current directory and specify it using this method of description.

Example for Using "#include"

If the specified file cannot be found in any directory searched, ICC740 outputs an include error.



1.9.3 Macro Definition

Use the "#define identifier" for character string replacement and macro definition. Normally use uppercase letters for this identifier to discriminate it from variables and functions.
This section explains how to define a macro and cancel a macro definition.

Defining a Constant

A constant can be assigned a name. This provides an effective means of using definitions in common to eliminate magic numbers (immediate with unknown meanings) in the program.

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

Defines that the threshold = 100.

Sets the upper limit at +50.

Sets the lower limit at -50.

Defining a Character String

A string can be assigned a name.

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

The defined character string is inserted at the position of "TITLE".

Defining a Macro Function

The command "#define" can also be used to define a macro function. This macro function allows arguments and return values to be exchanged in the same way as with ordinary functions. Furthermore, since this function does not have the entry and exit processing that exists in ordinary functions, it is executed at higher speed.
What's more, a macro function does not require declaring the argument's data type.

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

Macro function that returns the argument's absolute value.

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

The symbol "\" denotes successive description. Descriptions entered even after line feed are assumed to be part of a continuous character string.

Enclose a complex statement with brackets '{' and '}'.

Canceling Definition

```
#undef      identifier
```

Replacement of the identifier defined in "#define" is not performed after "#undef". However, do not use "#undef" for the following four identifiers because they are the compiler's reserved words.

- _FILE_ ; Source file name
- _LINE_ ; Line number of current source file
- _DATE_ ; Compilation date
- _TIME_ ; Compilation time
- _IAR_SYSTEMS_ICC_ ; ICC compiler identifier
- _STDC_ ; ICC compiler identifier
- _TID_ ; Target identifier
- _VER_ ; Compiler version number

1.9.4 Conditional Compile

ICC740 allows you to control compilation under three conditions.
Use this facility when, for example, controlling function switchover between specifications or controlling incorporation of debug functions.
This section explains types of conditional compilation and how to write such statements.

Various Conditional Compilation

The following lists the types of conditional compilation that can be used in ICC740.

Description	Content
#if Condition expression A #else B #endif	If the condition expression is true (not 0), ICC740 compiles block A; if false, it compiles block B.
#ifdef identifier A #else B #endif	If an identifier is defined, ICC740 compiles block A; if not defined, it compiles block B.
#ifndef identifier A #else B #endif	If an identifier is not defined, ICC740 compiles block A; if defined, it compiles block B.

In all of these three types, the "#else" block can be omitted. If classification into three or more blocks is required, use "#elif" to add conditions.

Specifying Identifier Definition

To specify the definition of an identifier, use "#define" or ICC740 compiler option '-D'.

```
#define identifier          ← Specification of definition by "#define"
%ICC740 -D identifier       ← Specification of definition by compiler option
```

Example for Conditional Compile Description

The following shows an example for using conditional compilation to control incorporation of debug functions.

```
#define    DEBUG

void    main(void)
{
            ¦
            ¦
#ifdef    DEBUG
      check_output();
#else
      output();
#endif
            ¦
            ¦
}

#ifdef    DEBUG
void    check_output(void)
{
            ¦
            ¦
}
#endif
```

It defines an identifier "DEBUG". (Set to debug mode.)

When in debug mode, it calls "debug function;" otherwise, it calls "ordinary output function". In this case, it calls "debug function".

When in debug mode, it incorporates "debug function".

Chapter 2
Downloading a Program into the ROM

This chapter describes the precautions to be taken when creating a built-in program centering round the extended functions of the ICC740.

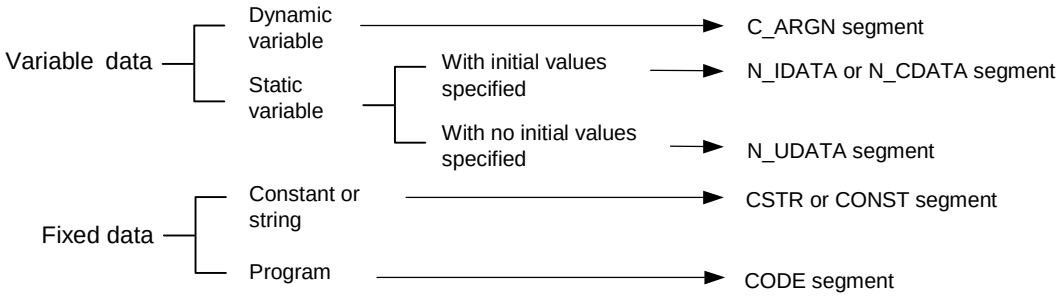
2.1 Memory Allocation

2.1.1 Types of Codes and Data

The codes and data that comprise a program come in various types, including those that can or cannot be rewritten and those that have or do not have initial values. All codes and data must be located in the ROM, RAM or stack areas according to their properties.
This section describes the types of codes and data that are generated by the ICC740.

Codes and data generated by the ICC740

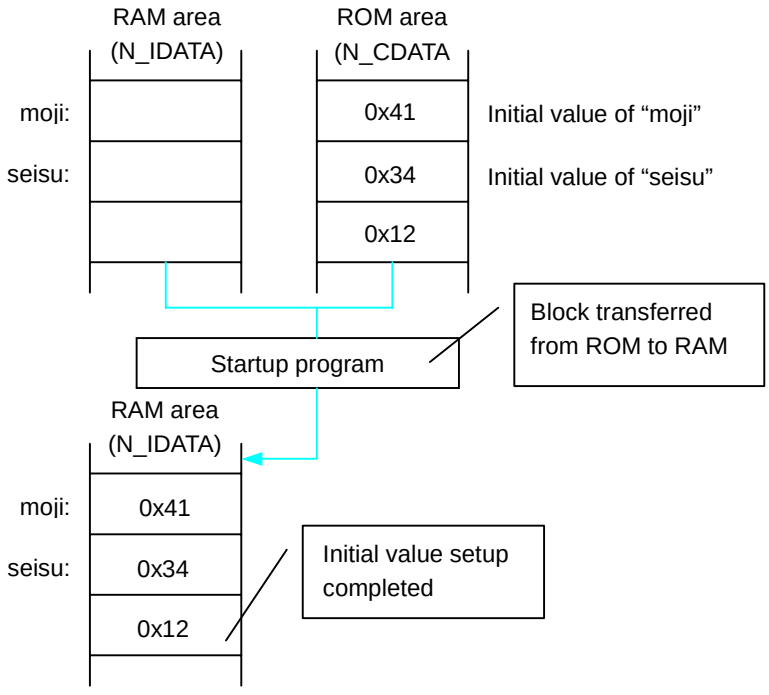
The following shows the types of codes and data generated by the ICC740 and the areas in which they are located.



Handling of static variables with initial values

The “static variables with specified initial values” are rewritable data and must therefore be located in RAM. However, if they are located in RAM, initial values cannot be set for them.
The ICC740 reserves an area for the static variables that come with specified initial values in RAM and stores their initial values in ROM. Then, in a startup program, it copies the initial values in ROM to the reserved area in RAM.

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



2.1.2 Segments Managed by the ICC740

The ICC740 manages the areas in which data or codes have been located as "segments."
This section describes the types of segments generated and managed by the ICC740 and how they are managed.

Segment configuration

The ICC740 classifies data by types as it manages segments. The configuration of segments managed by the ICC740 is shown below.

Segment name	Content	Location	
BITVARS	Static bit variable storage	RAM	Z
ZPAGE	Library variables (located in zero page)	RAM	Z
C_ARGZ	Stores Auto variables and arguments to functions (variables using the **zpage** keyword)	RAM	Z
Z_UDATA	Stores external variables without specified initial values (variables using the **zpage** keyword)	RAM	Z
Z_IDATA	Stores external variables with specified initial values (variables using the **zpage** keyword)	RAM	Z
EXPR_STACK	Expression stack (whose size is specified in lnk740.xcl, manipulated by register X)	RAM	Z
INT_EXPR_STACK	Interrupt expression stack (whose size is specified in lnk740.xcl, manipulated by register X)	RAM	Z
CSTACK	Ordinary stack (whose size is specified in lnk740.xcl, with location specified in **cstartup**)	RAM	Z/N
NPAGE	Library variables (located in other than zero page)	RAM	N
C_ARGN	Stores Auto variables and arguments to functions	RAM	N
N_UDATA	Stores external variables without specified initial values	RAM	N
N_IDATA	Stores external variables with specified initial values	RAM	N
ECSTR	Writable character strings, for which area is reserved when the ICC740 option -y is specified	RAM	N
RF_STACK	Stack used during recursive calls (whose size normally is 0; 256 bytes or more is reserved when used)	RAM	N
RCODE	Stores library code	ROM	-
Z_CDATA	Stores initialization constants (initial values for the Z_IDATA segment)	ROM	-
N_CDATA	Stores initialization constants (initial values for the N_IDATA segment)	ROM	-
C_ICALL	Table for indirect function calls	ROM	-
C_RECFN	Table for recursive functions	ROM	-
CSTR	Stores constants and strings	ROM	-
CCSTR	Stores initial values when the ICC740 option -y is specified	ROM	-
CODE	Stores a program	ROM	-
CONST	Stores constants	ROM	-
C_FNT	Special page jump table	ROM	-
INTVEC	Interrupt vector table	ROM	-

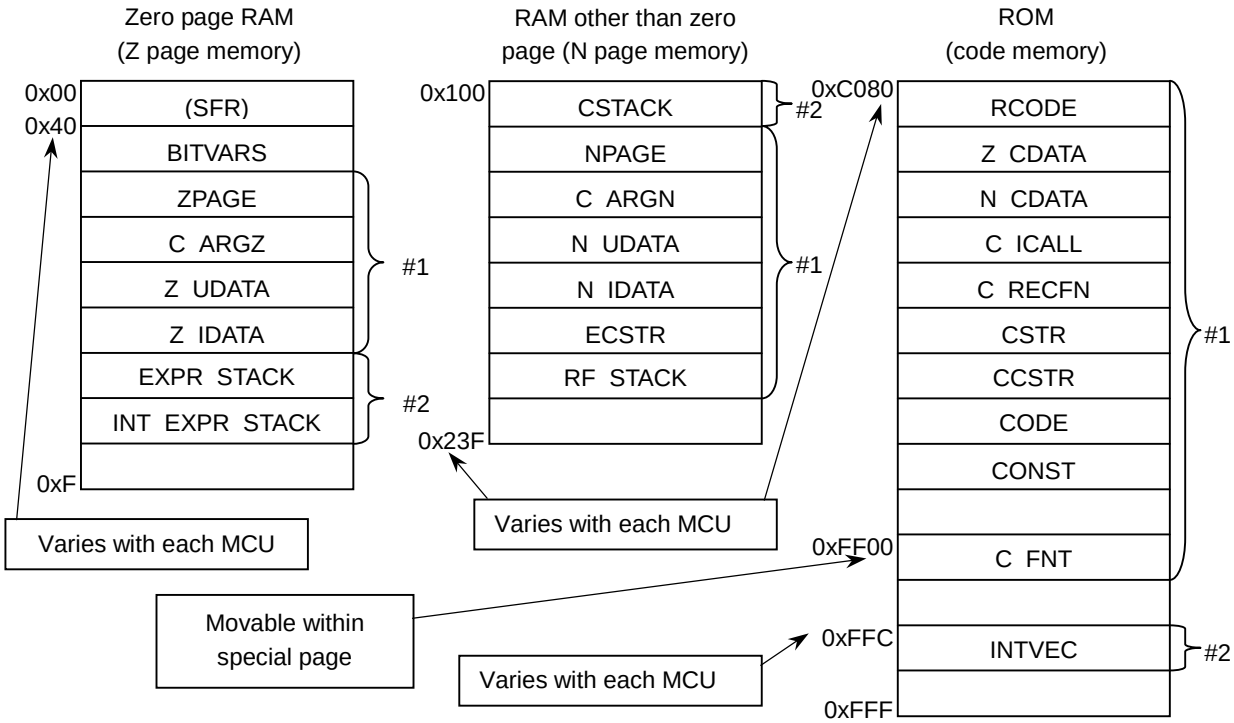
*Z: Located in zero page
*N: Located in other than zero page
*Z/N: Located in zero page or page 1

2.1.3 Controlling Memory Allocation

The ICC740 allows segments to be located in memory efficiently in a manner suitable for the user system.

Segment locations in memory

The following shows how segments will be located in memory depending on data types.



The sizes marked by #1 are set by the linker XLINK. The sizes marked by #2 are set in the lnk740.xcl file.

Adding segment names (“lnk740.xcl”)

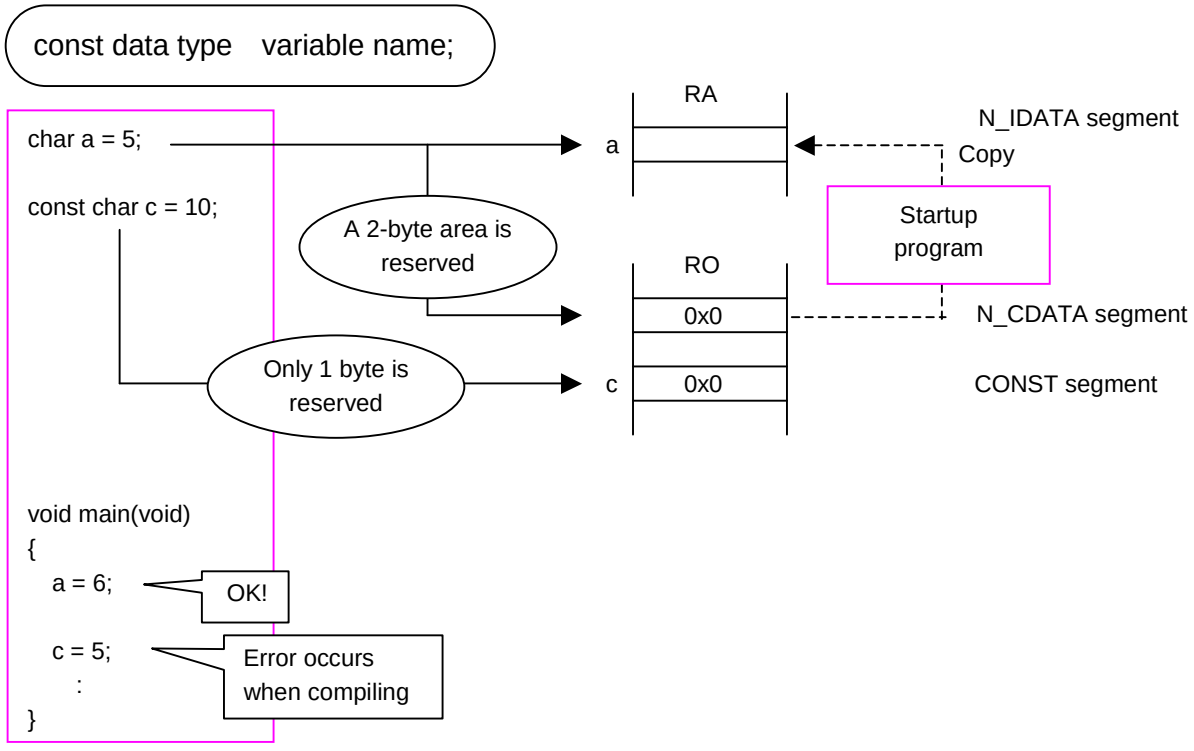
The segments generated by the ICC740 have the order specified in the link command file “lnk740.xcl” in which they are to be located. The newly created segments must each have a segment name added in the link command file “lnk740.xcl.”

```
-!                    - lnk740.xcl -

        XLINK 4.44, or higher, command file to be used with the 740
        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
                                                          .
                                                          .
                                                          .
```

Add a segment name for the newly created segment

Forcibly locating segments in ROM (const qualifier)

If a variable has its initial value specified during type declaration, both RAM and ROM areas are reserved. However, if this variable is of a fixed value that does not change during program execution, you may write the “const qualifier” when declaring its type. In that case, only a ROM area is reserved, with no RAM areas used, helping to save on the amount of memory needed. Furthermore, because explicit assignments are checked when compiling, it is possible to inspect whether unrewritable areas are rewritten.



Column Changing segment names (#pragma codeseg)

#pragma codeseg (segment name to be changed)

Change the name of a program segment generated by the ICC740.

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

#pragma codeseg ( NEW_CODE )

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

Expansion image

```
    RSEG CODE
func1:
    . . .

    RSEG
NEW_CODE
func2:
    . . .
```

func1 is expanded with the default segment name, whereas func2 is expanded with a changed segment name. The new segment name with which to be changed must not be a duplicate of any reserved segment name of the compiler.

In addition to the above, there are following cases in which a segment name is changed (those that are located in a named segment).

#pragma memory=constseg (segment name) Locates a variable in the named segment

Example

```
#pragma memory=constseg ( TABLE )
char ar[] = { 1, 3, -1, 5, -3 } ;      ← Places the constant array arr in the ROM segment "TABLE"
#pragma memory=default                 ← Restores memory allocation to the default area
```

Note that if this constant array is to be accessed in another module, an equivalent declaration must be written in that module too.

#pragma memory=dataseg (segment name) Locates a variable in the named segment

Example

```
#pragma memory=dataseg ( USRDAT )   ← Places three variables in the RAM area named "USRDAT"
char USRDAT_data1 ;
char USRDAT_data2 ;
int USRDAT_data3 ;
#pragma memory=default
```

Note that if these variables are to be accessed from another module, an equivalent extern declaration must be written.

#pragma memory=zpage Locates a variable in the Z page area (0x00–0xFF)

Example

```
#pragma memory=zpage
int buf[5] ;                        ← Places the variables buf and f in ZPAGE memory
float f ;
no_init char *str[5] ;              ← Locates the variable str forcibly in NO_INIT memory
#pragma memory=default
int a ;                             ← Locates the variable a in the DATA area
```

2.2 Initialization Setup Files

2.2.1 Roles of the Initialization Setup Files

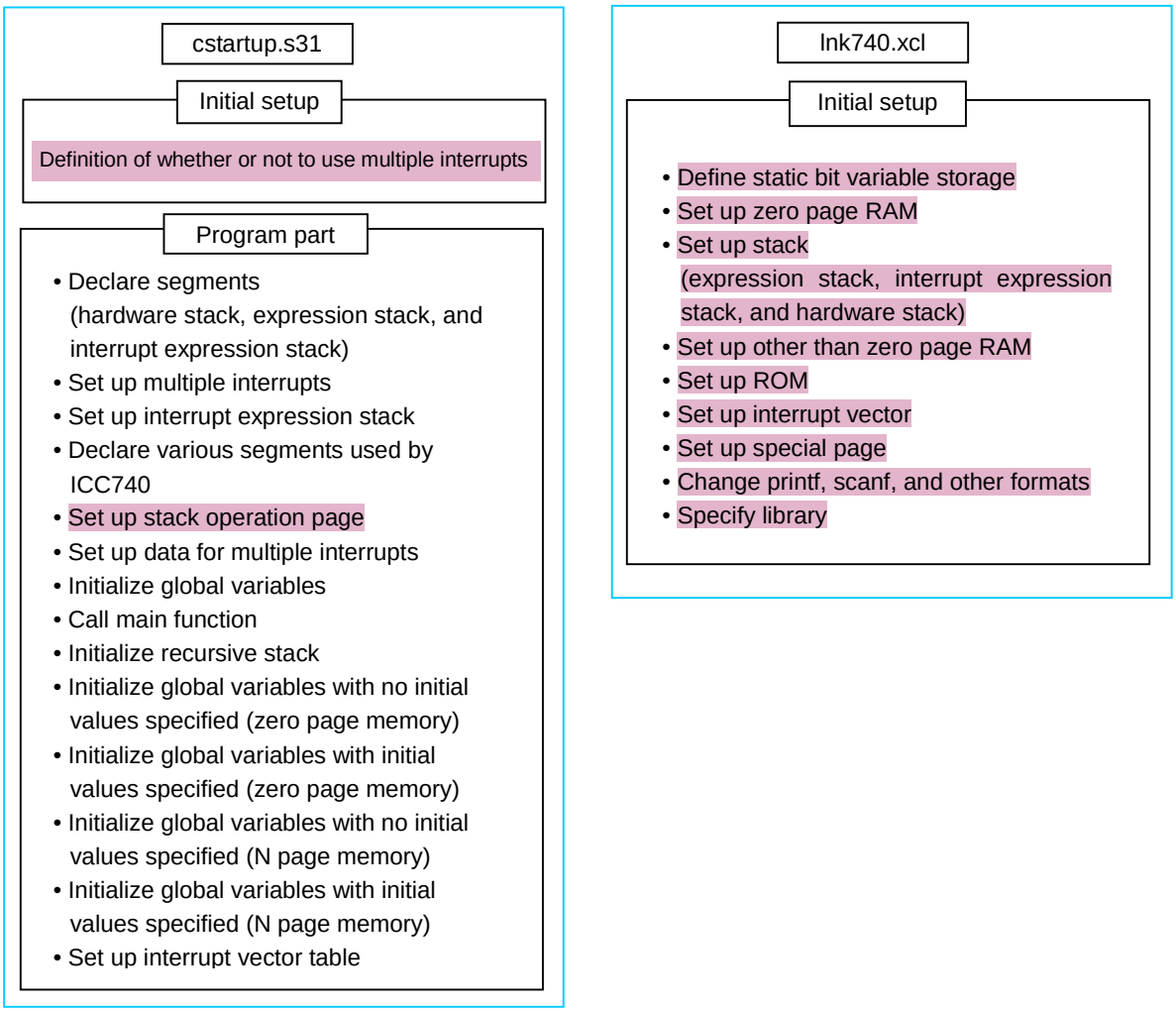
The ICC740 has two initialization setup files, namely the startup program "cstartup.s31" and the link command file "lnk740.xcl."
This section describes the roles and structures of the initialization setup files.

Roles of the initialization setup files

The following lists the roles of the initialization setup files:
(1) Reserve the stack area
(2) Initialize the microcomputer
(3) Initialize the static variable area
(4) Call the main function
(5) Set up the interrupt vector table

Structures of the initialization setup files

The following shows the structures of the initialization setup files.



* The shaded definitions and settings are detailed in Sections 2.2.2 and 2.2.3.

2.2.2 Startup Program

For a built-in type program to be run normally, it is necessary to initialize the microcomputer and set up the stack area before the program can be processed. Normally, these types of processing cannot be written in C language. Therefore, a program for initial setup is written in assembly language, separately from the C language source program. This is the "startup program."
The startup program "cstartup.s31" available with the ICC740 is described in this and subsequent sections.

Modifying the startup program

Change the following parts of the startup program to make it suitable for the user program to be created.

cstartup.s31

- Definition of whether or not to use multiple interrupts
- Stack page settings
- Interrupt vector registration
- Reset vector registration

Definition of whether or not to use multiple interrupts ("cstartup.s31")

If multiple interrupts are not used, memory usage can be reduced 1 byte for RAM and 4 bytes for ROM by defining a NO_INTERRUPTABLE_ISR segment. Where and how to define it is shown below.

```
;------------------------------------------------------------------
; 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
; be included.
;------------------------------------------------------------------
;#define NO_INTERRUPTABLE_ISR
```

When not using multiple interrupts, remove the semicolon (;) at the beginning of the line.
To use multiple interrupts, do not remove the semicolon (;) at the beginning of the line.

Setting up the stack page (“cstartup.s31”)

Set up the stack page. Make sure the value of each bit in the CPU mode register is set to suit the operating environment. Where and how to write is shown below.

```
;------------------------------------------------------------------;
;   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
```

Set up the stack page. By default, it is set in page 1.
If it needs to be set in page 0, set up as shown below (for the 3803 group’s case):

```
LDM   #08H, 3BH
```

Registering the interrupt vector (“cstartup.s31”)

By altering the content of the INTVEC segment in the cstartup.s31 included with the ICC740, register the interrupt processing functions as suitable for the microcomputer used. For this purpose, rewrite the lines 352–372 as shown below.

```
    COMMON   INTVEC

?CSTARTUP_INTVEC:
          BLKB      0FFFEH - 0FFD
#if 0
#if defined(MELPS_37600)
          BLKB      40H - 6         ;
#elif defined(MELPS_MULDIV)
          BLKB      20H - 4         ;
#else
          BLKB      20H - 2         ;
#endif
#endif
?CSTARTUP_RESETVEC:
          WORD      init_C
          ENDMOD    init_C
```

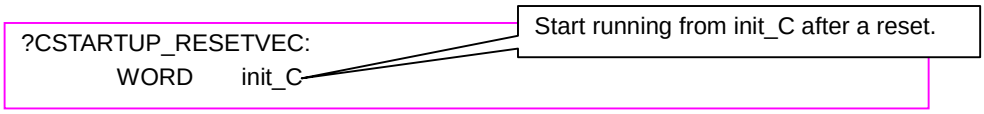
```
          EXTERN Int2, Timer1, Int0
          COMMON   INTVEC

?CSTARTUP_INTVEC:
          WORD      init_C          ; +0x00 : BRK
          WORD      init_C          ; +0x02 : AD_SIO3T
          WORD      init_C          ; +0x04 : INT4_CNTR2
          WORD      init_C          ; +0x06 : INT3
          WORD      Int2            ; +0x08 : INT2
          WORD      init_C          ; +0x0a : SIO2_TimerZ
          WORD      init_C          ; +0x0c : CNTR1_SIO3R
          WORD      init_C          ; +0x0e : CNTR0
          WORD      init_C          ; +0x10 : Timer2
          WORD      Timer1           ; +0x12 : Timer1
          WORD      init_C          ; +0x14 : TimerY
          WORD      init_C          ; +0x16 : TimerX
          WORD      init_C          ; +0x18 : SIO1T
          WORD      init_C          ; +0x1a : SIO1R
          WORD      init_C          ; +0x1c : INT1
          WORD      Int0            ; +0x1e : INT0_TimerZ
?CSTARTUP_RESETVEC:
          WORD      init_C          ; +0x20 : reset
          ENDMOD    init_C
```

Example for the 3803 group (varies with each microcomputer used)

If two or more interrupt sources are set at the same vector address, write them as one interrupt source. Note also that there is no need to add a reset because the reset vector is set at the bottom.
Declare the interrupt processing function to be externally referenced by using the directive command EXTERN. If there is any undefined interrupt vector, we recommend setting init_C in that vector as for a reset without leaving it blank, to prevent the program from running out of control.
Write the beginning address of the INTVEC segment in the lnk740.xcl file.

Registering the reset vector (“cstartup.s31”)

Set the reset vector (module name to be started after a reset).



2.2.3 Link Command File

The link command file is used to set up details about the memory map.
The following describes the link command file "lnk740.xcl."

Modifying the link command file

Change the following parts of the link command file to make it suitable for the user program to be created.

lnk740.xcl

- Segment location and beginning/ending address settings
- Stack size settings
- Interrupt vector beginning/ending address settings
- Special page beginning/ending address settings
- Library specification

Segment location and beginning/ending address settings (“lnk740.xcl”)

Set the locations in memory and the beginning/ending addresses of the segments generated by the ICC740. Segments with no beginning addresses specified are located at contiguous memory addresses following the previously defined segment. Segments separated by a comma are allocated to memory addresses in the order they are written.

```
-!   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
     "41-FF" address range.
     If you have the CSTACK (processor stack) segment outside Z
     you have to give it an address and XLINK will no longer try
     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 800

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

Define the BITVARS segment.
Specify its location by an address in bit units starting from the address 0.

Set up the zero page RAM.
Specify segments to be located in zero page memory.

Set up other than the zero page RAM.
Specify segments to be located in N page memory.
If CSTACK is set in page 1, NPAGE is allocated starting from the address following CSTACK.

Set up ROM.
Specify segments to be located in ROM and their addresses (except the reserved area and interrupt vector area).

* -Z(XXX): -Z is used to specify a segment location. XXX assigns type to the segment. The following types are used in this link command file:

BIT	Bit memory
ZPAGE	Zero page data memory
NPAGE	Data memory accessed by specifying an absolute address
CODE	Code memory

* Segment name = YY (-ZZ): The segment is allocated in such a way that it will start from the beginning address YY of the specified range. (ZZ specifies the ending address.) If no address ranges need to be specified, write only the beginning address.

Setting the stack size (“lnk740.xcl”)

Those that use a stack include a temporary area used in complicated calculations and a return address. In the ICC740, stacks are classified as shown below, for the purpose of efficient memory usage.

CSTACK	Stack. Holds a hardware stack.
EXPR_STACK	Expression stack. Temporarily holds a result while an expression is being evaluated in normal processing.
INT_EXPR_STACK	Interrupt expression stack. Temporarily holds a result while an expression is being evaluated in interrupt processing.
RF_STACK	Recursive stack. Stores a local variable and parameters for the enclosed call of a recursive function.

Assign the stack an appropriate size. An excessively small stack size may cause the system to go wild, and an excessively large stack size may lead to a wasteful use of memory.
The following describes how to set the size of each stack: the expression stack, the interrupt expression stack, and the C stack. These stacks must have their necessary sizes set in this link command file “lnk740.xcl.”

```
-!   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 0 below.
       You must give this stack space if you have C written interrupts that
       need an expression stack.    -!

-Z(ZPAGE)INT_EXPR_STACK+0


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

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

Set up an expression stack segment. By default, its size is set to 20h.

Set up an interrupt expression stack segment. By default, its size is set to 0h. If expressions are to be evaluated in interrupt processing, set the necessary size of this stack segment.

Set up a C stack segment. By default, its size is set to 100h through 13Fh.

* Segment name + YY: The segment is allocated in such a way that it will have the set memory size (YYh bytes).
* To locate a C stack in page 0, alter the file description as shown below.

  -Z(ZPAGE) + 40

  40h bytes of space is allocated after INT_EXPR_STACK.

  *cstartup.s31 must also be altered.

Setting the beginning/ending addresses of the interrupt vector (“lnk740.xcl”)

Set the beginning/ending addresses of the interrupt vector. An example of how to set is shown below.

```
-!    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
```

Set up an interrupt vector segment.
Specify an address range from the beginning of the interrupt vector to the end of the reset vector.

Setting the beginning/ending addresses of the special page (“lnk740.xcl”)

Set the beginning/ending addresses of the special page. An example of how to set is shown below.

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

Set up a special page segment.
Specify the area for a function using the extended statement tiny_func.

Specifying a library (“lnk740.xcl”)

Specify a library. An example of how to specify is shown below.

```
-! 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   -!
```

Specify a library.
Select the necessary library and add the -C option when specifying it.

* The library contains a default cstartup module. Unless the -C option is added, this default module will be used.
* Type of library
  - cl7400t: Tiny model
  - cl7400l: Large model
  - cl7401t: Tiny model for microcomputers without MUL/DIV
  - cl7401l: Large model for microcomputers without MUL/DIV
  - cl7402t: Tiny model for microcomputers with extended memory access
  - cl7402l: Large model for microcomputers with extended memory access

2.3 Extended Functions for Putting into the ROM

2.3.1 Variable Location

The accessible memory space of the 740 family microcomputers is 64 Kbytes at maximum. The ICC740 manages this memory space by dividing it into the "Z page area" at the addresses 0000h–00FFh and the "N page area" at the addresses 0100h–FFFFh for management purposes.
This section describes how to locate variables and functions in these areas and how to access.

Z page area and N page area

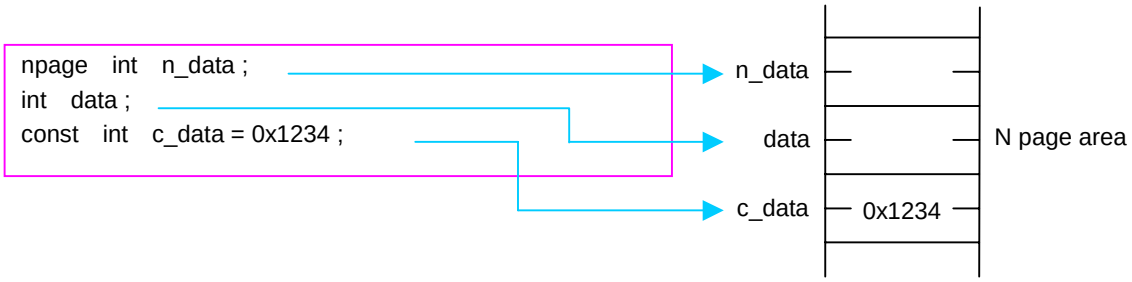
The ICC740 manages up to 64 Kbytes of access space in two separate areas: the "Z page area" and the "N page area." The features of each area are summarized below.

Area name	Content
Z page area	A space in which the 740 family microcomputers can access data efficiently at high speed. It is a 256-byte area at the absolute addresses 0000h–00FFh, in which the stacks and internal RAM are located.
N page area	A 65,280-byte area at the absolute addresses 0100h–FFFFh that the 740 family microcomputers can access, in which the stacks, internal RAM, and internal ROM are located.

Location of variables

type specifier variable name;

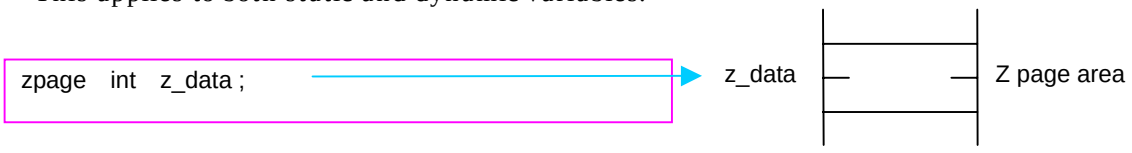
Normally, RAM data is located by default in the N page area. The data which has had npage or zpage specified during type declaration and the ROM data which has had the const qualifier specified are located in the N page area.
This applies to both static and dynamic variables.



Locating variables in the zpage area

zpage type specifier variable name;

When zpage is specified during type declaration, RAM data is located in the Z page area.
This applies to both static and dynamic variables.

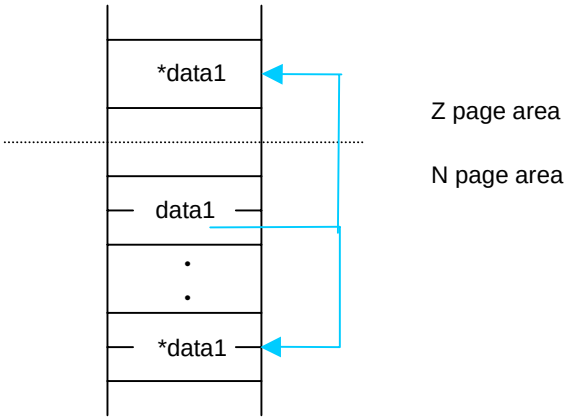


Pointer variables

A pointer variable itself and its reference address both are located in the default area (N page area). Also, it is possible to specify the default area (N page area) for the pointer variable itself and the Z page area for the reference address.

type specifier *variable name;

```
int *data1 ;
```



Specifying zpage for pointer variables

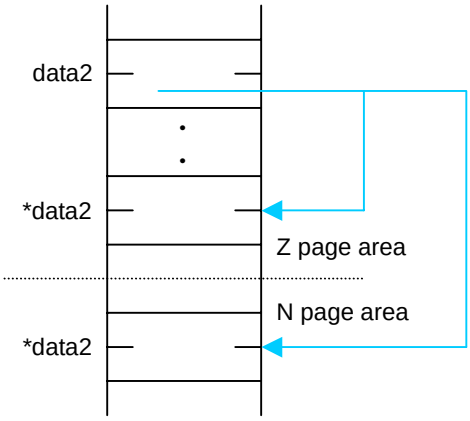
By specifying zpage for a pointer, you can specify the size of the address to be stored in it and the area in which the pointer itself is to be located.

(1) When you specify the area in which the pointer itself is to be located

The pointer itself is located in the specified area (Z page area), and the reference address is located in the Z page area or the N page area depending on the data.

type specifier *zpage variable name;

```
int *zpage data2 ;
```

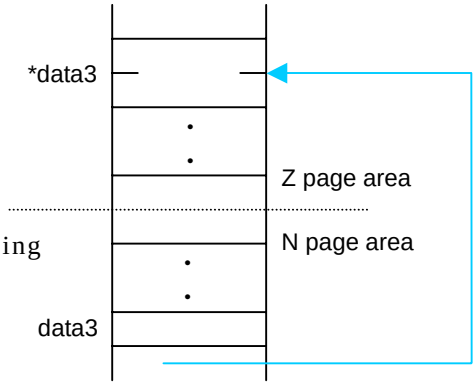


(2) When you specify the area in which the reference address is to be located

The reference address is located in the specified area (Z page area), and the pointer itself is located in the default area (N page area).

type specifier zpage *variable name;

```
int zpage *data3 ;
```



However, because this statement will cause a warning

Warning [w23]: Cannot represent pointer type

make sure that only the Z page area is referenced.

2.3.2 Handling of Bits

The ICC740 allows data to be handled in bit units. This is accomplished by using a bit variable in two ways: one using extended feature, and the other using a union.
This section describes each method.

Bit variables

The following shows how to handle bits in the ICC740.

(1) When using the single bit variable "bit" of the extended language feature
The bit variable represents one bit in a Z page sfr variable.

```
sfr   Port1 = 0x02 ;      // Defines the address 0x02 of the SFR as "Port1" //
bit   Port14 = Port1.4 ;// Defines the 4th bit of "Port1" as "Port14" //
bit   Port15 = 0x02.5 ; // Defines the 5th bit at the address 0002h of the SFR as "Port15" //
```

Instructions using SEB and CLB are generated.
Be aware that this bit variable is different from the C standard bit field.

(2) When using a union
The union represents a bit type structure and a byte variable.
It can be used in both Z and N pages.

```
typedef union{
    unsigned char     byte ;     // Used for byte access //
    struct{                      // Used for bit access //
        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 ;
}
```

Instructions corresponding to Z and N pages respectively are generated.

Column About the location of bit-fields

If it is desired to change the order in which bit-fields are stored, use the #pragma directive. Because the order in which bit-fields are stored depends on the compiler, this method helps to avoid the portability problems.

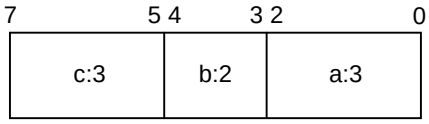
#pragma bitfields=default

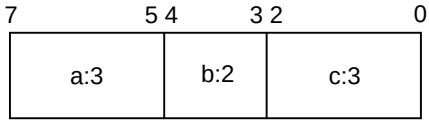
Restores the original bit-field storage sequence

#pragma bitfields=reversed

Reverses the bit-field storage sequence

```
struct{
        short   a : 3 ;
        short   b : 2 ;
        short   c : 3 ;
}bit1 ;

#pragma bitfields=reversed

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





2.3.3 Control of the I/O Interface

To control I/O interfaces in an embedded system, specify absolute addresses for variables. There are two methods to specify absolute addresses in the ICC740: one by defining the SFR area, and one by using a pointer.
This section describes each method.

Defining the SFR area

Use the sfr variable of the extended language feature to set the SFR area. This SFR setting should normally be prepared as a separate file, which can then be included in the source program.
An example of a SFR area definition file is shown below.

SFR area definition file <io3803.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 ;
      :
```

Absolute address settings

<Source file>

```
#include "io3803.h"
              :
void main( void )
{
      :
      TX   = 0x94;
      TM.3 = 0 ;
      :
}
```

Loads the SFR area definition file

References the SFR area for byte access

References the SFR area for bit access

Specifying an absolute address by a pointer

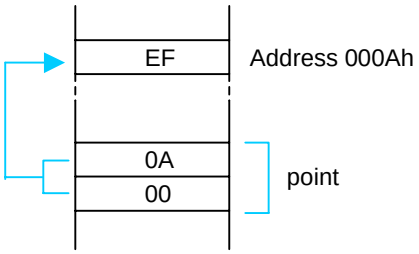
An absolute address can be specified by using a pointer. An example of how to specify is shown below.

Example: Assign 0xEF to the address 000Ah

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

|| When put together in one line

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



2.3.4 Alternative Way when Unable to Write in C Language

Hardware related processing when "the processing time is insufficient" or "to control the C flag directly" cannot always be written in C language. In such a case, the ICC740 allows assembly language to be written directly into the C language source program (known as the "inline assemble feature). The inline assemble feature can be accomplished using the "inline function" or the "asm function."
This section describes each method.

Generating assembler instructions (inline function)

There are following seven inline functions:

break_instruction()	:	Generates the BRK instruction and then the NOP instruction
cld_instruction()	:	Generates the CLD instruction
disable_interrupt()	:	Generates the SEI instruction
enable_interrupt()	:	Generates the CLI instruction
enter_stop_mode()	:	Generates the STP instruction
enter_wait_mode()	:	Generates the WIT instruction
nop_instruction()	:	Generates the NOP instruction

An example of how to write is shown below.

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

Expanded image

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

* When using inline functions, be sure to include intr740.h. This header file exists in the inc folder in the folder in which you installed the ICC740.

Writing only one line in assembly language (asm function)

```
__asm ("string");
```

When written as shown above, the string enclosed in double-quotes (") is expanded into an assembly language source program directly as is (including the space and tab).
Because this statement can be written in either the inside or outside of a function, it may prove useful when the flags or registers need to be manipulated directly or high-speed processing is desired. An example of how to write is shown below.

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

* To write __, enter two consecutive underscores (_).

2.4 Linkage with Assembly Language

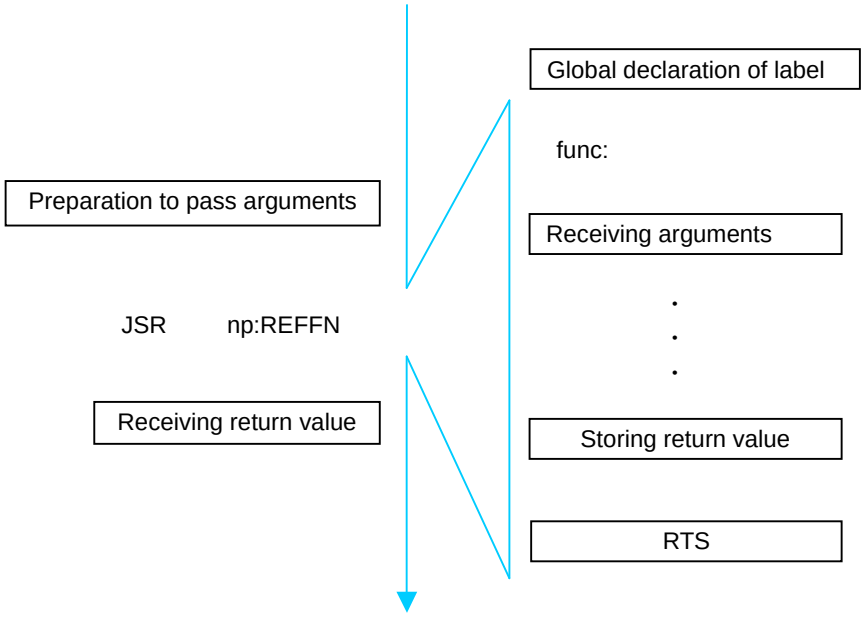
2.4.1 Interfacing between Functions

Assembly language subroutines can be called from a C language program. Use this interface to call assembly language from C language.
This section describes interfacing between functions in the ICC740.

Entry and exit processing of functions

The primary processing needed to call a function in the ICC740 consists of the following two:
(1) Argument transfer to and from function
(2) Return value transfer to and from function

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



Rules for passing arguments

The ICC740 uses different places in which to store arguments depending on the data types of arguments. The following shows how arguments are passed to a function.

Type of argument	Passed via
char type	Accumulator *
Other types	C_ARGN or C_ARGZ

* For the first argument only

Rules for passing return values

The ICC740 uses different places in which to store return values depending on the data types of return values. The following shows how return values are passed to a function.

Data type	Returned via
char type	Accumulator *
Other types	EXPR_STACK

Types of stacks

The ICC740 stores the arguments and return address passed to when calling a function in the segments shown below. Therefore, the ICC740 does not construct a stack frame.

	Segment
Argument	C_ARGN (other than zero page) C_ARGZ (zero page)
Return address	CSTACK

Rules for converting function symbols into assembly language

The ICC740 uses the same method to convert symbols irrespective of the properties of functions. The symbol conversion rule is shown below

Function name	Assembler symbol name
Example func()	Function name: Example func:

To declare the functions in an assembly language program to be called from a C language program, use the assembler interface keyword "DEFFN." For the functions in an assembly language program to be referenced from a C language program, you need to declare the names of those functions with "PUBLIC." Then, in the C language program, declare those functions names as "EXTERN."

```
DEFFN    function name ( a, 0, b, 0, 0x8000+x, 0, y, 0 )
```

a: auto variable size in zero page of the function to be set in the C_ARGZ segment
b: auto variable size in other than zero page of the function to be set in the C_ARGN segment
x: Argument size in zero page of the function to be set in the C_ARGZ segment
y: Argument size in other than zero page of the function to be set in the C_ARGN segment

An example of how to write a program is shown below.

```
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
```

Save/restore register X and flag

b.s31

* The arguments a, b, x, and y must be 0 because the assembler functions do not have arguments.
* The keyword DEFFN is needed by the linker to calculate the sizes of the segments C_ARGZ and C_ARGN.

2.4.2 Calling Assembly Language from C Language

This section describes how to write a program to call assembly language subroutines as C language functions.

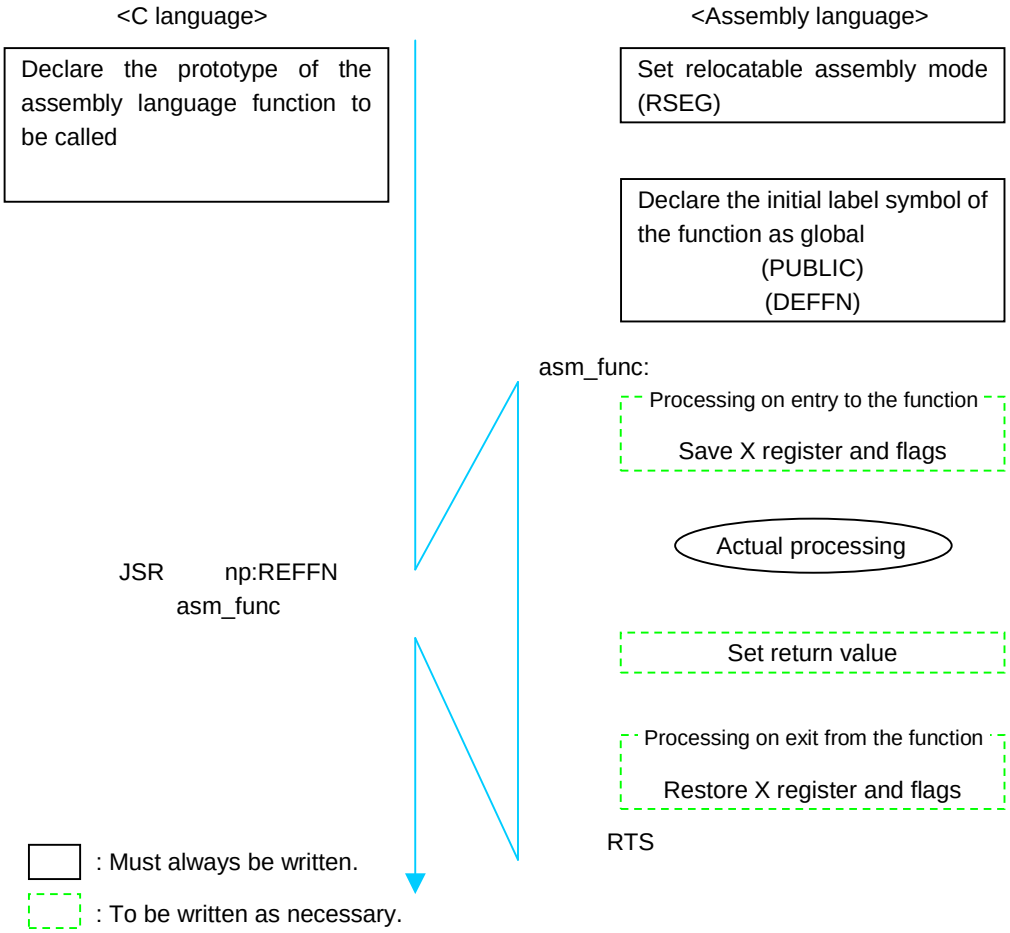
Calling assembly language subroutines

To call assembly language subroutines (assembly language functions) from a C language program, follow the rules described below.

(1) Write the subroutine in a separate file from the C language program.
(2) For the subroutine name, follow the symbol conversion rules.
(3) In the C language program from which to call, declare the prototype of the subroutine (assembly language function). At this time, use the storage-class specifier "extern" to declare the subroutine as externally referenced.
(4) In the subroutine (assembly language function), do not normally alter the X register and flag values that are used exclusively by the ICC740. If the X register or flag value needs to be altered, save the value to the stack on entry to the function and restore it from the stack on exit from the function.

```
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
```



Example for calling a subroutine

The following shows a sample program that indicates the result of a count-up operation on LEDs. Create the LED display part in assembly language and the count-up part in C language, and then link.

<Count-up part>

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

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

      led() ;
}
```

<LED display part>

```
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 Interrupt Handling

The ICC740 allows interrupt handling to be written as C language functions. The procedure consists of the following four steps:

(1) Writing the interrupt handling function
(2) Setting the interrupt disable flag (I flag)
Do this by using an inline function.
(3) Registering to the interrupt vector area
(4) Setting up the interrupt vector segment

2.5.1 Example for Writing Interrupt Handling Functions

This section shows an example for writing a program that clears the content of the "counter" to 0 each time an INT0 interrupt (rising edge) occurs in the 3803 group and counts up the content of the "counter" each time an INT2 interrupt (falling edge) occurs.

Example for writing interrupt handling functions

An example of how to write the source file is shown below.

```
#include     <intr740.h>          /* Header file for inline function */
#include     "io3803.h"           /* SFR header file for the 3803 group */

unsigned char     counter ;

interrupt void Int0( void )     /* Interrupt handling function */
{
     cld_instruction() ;         /* CLD instruction to initialize decimal mode flag */
     counter = 0 ;
}

interrupt void Int2( void )     /* Interrupt handling function */
{
     cld_instruction() ;         /* CLD instruction to initialize decimal mode flag */
     if( counter < 9 ){
          counter++ ;
     } else {
          counter = 0 ;
     }
}

void main( void )
{
     /* (1) Set the corresponding interrupt enable bit to 0 (disabled) */
     ICON1.0 = 0 ;               /* INT0 interrupt enable bit → Disabled */
     ICON2.3 = 0 ;               /* INT2 interrupt enable bit → Disabled */

     /* (2) Set the interrupt edge select bit and interrupt source bit */
     INTEDGE.0 = 1 ;              /* INT0 asserted by a rising edge */
     INTEDGE.3 = 0 ;              /* INT2 asserted by a falling edge */
     INTSEL.0 = 0 ;              /* Interrupt source → INT0 interrupt */

     /* (3) Past one or more instructions, set the corresponding interrupt request bit to 0 (not
        requested) */
     nop_instruction() ;        /* Insert one instruction */
     IREQ1.0 = 0 ;               /* INT0 interrupt request bit → Not requested */
     IREQ2.3 = 0 ;               /* INT2 interrupt request bit → Not requested */

     /* (4) Set the corresponding interrupt enable bit to 1 (enabled) */
     ICON1.0 = 1 ;               /* INT0 interrupt enable bit → Enabled */
     ICON2.3 = 1 ;               /* INT2 interrupt enable bit → Enabled */

     enable_interrupt() ;       /* CLI instruction to enable interrupt */

     while( 1 ) ;                /* Interrupt wait loop */
}
```

If none of the decimal mode flags are used in the program, they do not need to be initialized in the interrupt handling function.

Defined in "io3803.h"

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

ICON1.0 denotes bit 0 of SFR ICON1.

Processor Status Register

(Automatically saved to the stack during interrupt)

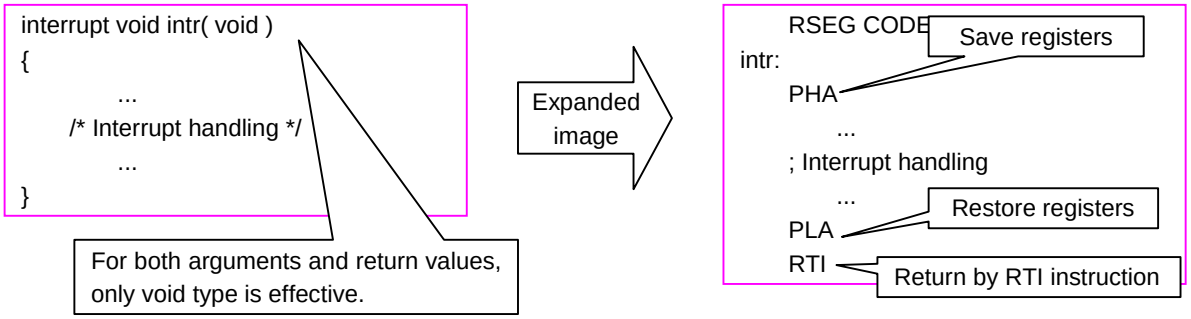
- D and T flags
  Initialized by init_C in cstartup.s31.
- I flag
  Set to 1 (disabled) immediately after the microcomputer is reset.

2.5.2 Writing Interrupt Handling Functions

The ICC740 allows interrupt handling functions to be written in C or assembly language.
This section describes the basic method for writing interrupt handling functions in C language.

Basic method for writing interrupt handling functions (C language)

Use the extended keyword interrupt to define interrupt handling functions. An example of how to write and an expanded image are shown below.



When written as shown above, the program saves and restores the 740 family registers and generates the RTI instruction, in addition to ordinary function procedures on entry and exit to and from the function. Note that the number of registers to be saved and restored varies depending on the content of the interrupt handling function concerned.

* The valid types of interrupt handling functions are only the void type, for both arguments and return values. All other types, if any declared, result in an error when compiled.

2.5.3 Setting the Interrupt Disable Flag (I Flag)

Setting the interrupt disable flag (I flag)

For interrupts to be generated, the interrupt disable flag (I flag) must be cleared to 0 (enabled). The ICC740 allows the I flag to be set by an inline function. To use inline functions, make sure "intr740.h" is included.

```
#include      <intr740.h>

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

Expanded image

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

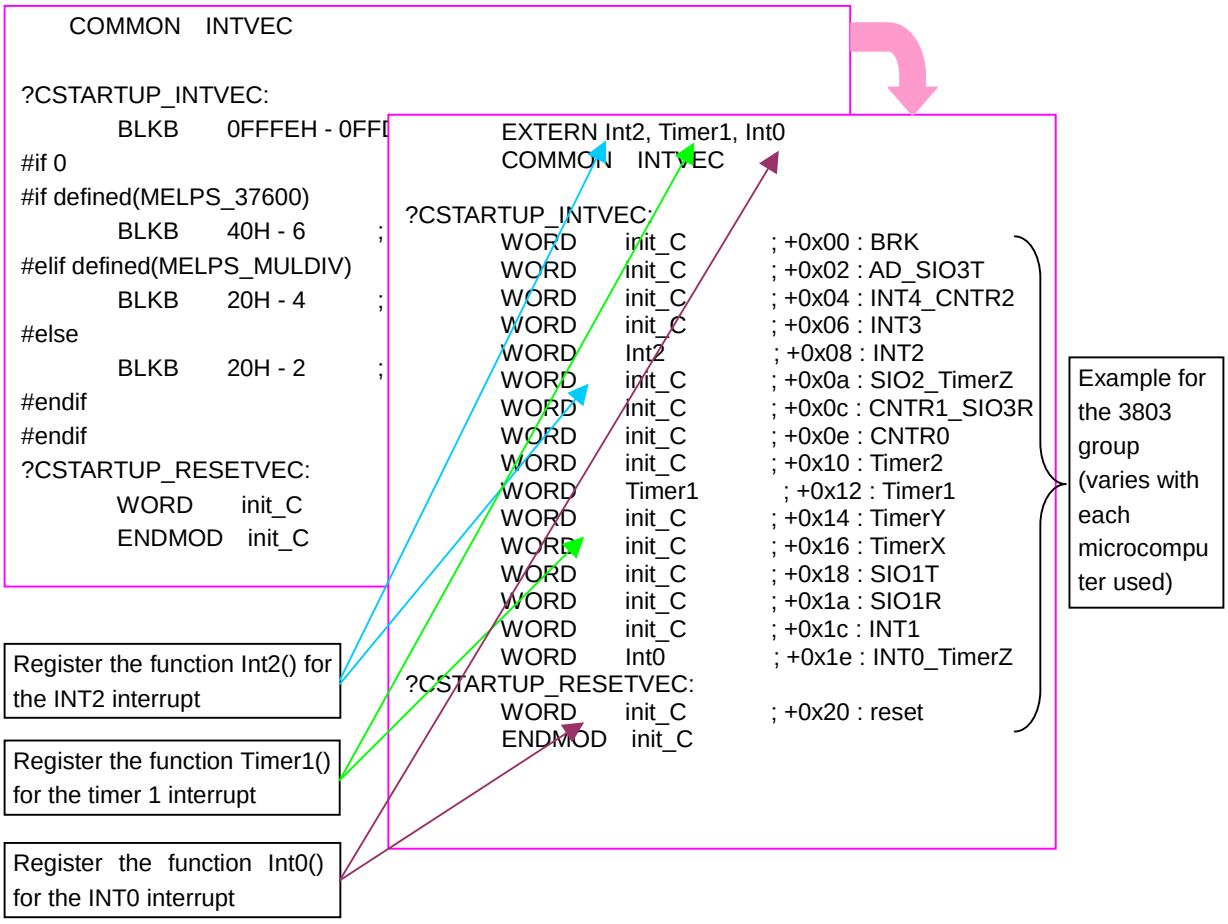
In the above example, enable_interrupt() and disable_interrupt() are replaced with the CLI instruction to clear the I flag to 0 and the SEI instruction to set the I flag to 1, respectively.

2.5.4 Registering to the Interrupt Vector Area

For interrupts to be used, it is necessary to write interrupt handling functions, as well as register the written interrupt handling functions in the interrupt vector area.
This section describes how to register the interrupt handling functions.

Registering interrupt handling functions

By altering the content of the INTVEC segment in the cstartup.s31 included with the ICC740, register the interrupt processing functions as suitable for the microcomputer used. For this purpose, rewrite the lines 352–372 as shown below.



The reset vector is set at the bottom. When the microcomputer is reset, the program starts from init_C that is registered in the reset vector. In this example, init_C is registered in the interrupt vectors for unused interrupts, the same way as for reset. There is no need to add a reset. The above WORD init_C line must be written as matched to the number of interrupt sources accommodated by the microcomputer used. init_C exists in the 134th line of cstartup.s31.

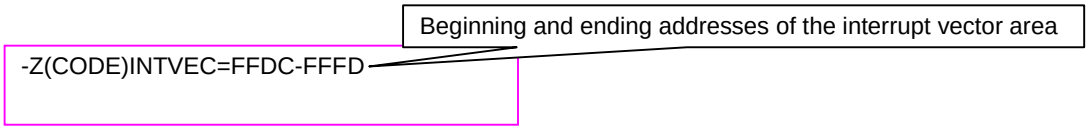
To register interrupt handling functions, follow the procedure described below.

(1) Declare the interrupt processing function to be externally referenced by using the directive command EXTERN.
(2) Register it to the interrupt vector.

2.5.5 Setting Up the Interrupt Vector Segment

Setting up the interrupt vector segment

To set up the interrupt vector segment, set the addresses given below in the interrupt vector segment “INTVEC” of the link command file “lnk740.xcl.”



* Make sure the beginning and ending addresses of the interrupt vector area you set here suit the microcomputer used.

REVISION HISTORY	740 family Programming Guide <C Language> Application Note

Rev.	Date	Description	
		Page	Summary
1.00	Sep 15, 2004	–	First edition issued

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.